Unit 11 Files

Structure:

11.1 Introduction

In the previous unit, you studied iostream hierarchy, standard IO stream library and its organization. You also studied the methods to format the output of the programs. In this unit you are going to study different methods of opening and closing a file, methods to check I/O status. You will also learn to deal with binary files. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or an array of uninterrupted bytes. For convenience, the "array" of bytes stored in a file is indexed from zero to *len*-1, where *len* is the total number of bytes in the entire file. You can open a file stream object if you can supply a file name along with an I/O mode parameter to the constructor when declaring an object. 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.

Objectives:

After studying this unit, you should be able to:

- describe opening and closing of files in different modes
- describe the ways to check the I/O status
- explain binary files
- discuss the important functions pertaining to files

11.2 Managing I/O Streams

In this section, we will discuss some useful functions. We will also see how to open a file to read and write simultaneously. We will also explore other ways to open a file and check if the opening was successful or not.

11.2.1 Opening and closing a file

For opening a file you have to create a file stream and link it to a filename. You can define a filename using ifstream, ofstream and fstream classes. These classes are contained in the header file fstream. Selection of classes which you are going to use depends on the purpose, i.e., whether you want to write data to the files or to read data from the files. There are two ways to open a file. These are:

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

1. Opening file using constructors

You already know that a constructor initializes an object when it is created. Here to initialize the file stream object, a filename is used. The following steps are required to do this:

Create a file stream object to manage the stream using appropriate class. That is, to create input stream, use ifstream class, and use ofstream class to create output stream.

Use any desired filename to initialize the object.

For example, the statement below opens the filename “results” for output:

```
ofstream outfile(“results”);     //output only
```

The outfile is created as an ofstream object that manages the output stream. This object can be any valid C++ name. The file ‘results’ is also opened by this statement. and this attaches it to the output stream outfile.

Similarly, the statement shown below declares infile as an ifstream object and attaches it to the data for reading (input).

```
iftream in file(“data”);          //input only
```

The following statements can be there in the program:

```
outfile << “total”;
outfile << sum;
infile << number;
infile << string;
```

You can also use the same file for reading and writing the data as shown in the example below:

```
Program1
…………..
………….
ofstream outfile(“salary”);       // the outfile is created and “salary’ is
                                          connected to it
…………..
………….
Program2
…………..
………….
ifstream infile(“salary”);        //infile is created and ‘salary” is
connected to
it.
```

When the stream object expires, then connection with the first file is closed automatically. As you can observe from the above statements, when program1 terminates, the salary file is disconnected from the outfile stream. When the program2 is terminated a similar action is taken.

We can use one program instead of two programs (one for writing data and another for reading data). For example:

```
………….
………….
 outfile.close();           //disconnects salary from outfile and connects
to infile
ifstream infile(“salary”);
…………
…………
infile.close();             //disconnect salary form infile
```

You can observe that even though one program is used, two filestream objects are created. One is an outfile to put data to the file, and another, an infile to get data from the file. The statement outfile.close(); disconnects the file from the output stream outfile. Note that the object outfile continues to exist, and later the salary file can again be connected to outfile or to any other stream. In the example shown above we have connected the salary file to the infile stream to read data.

The program below uses a single file for both writing and reading the data. First it takes data from the keyboard, and writes it to the file. The file is closed once the writing is completed. The program again opens the same file, reads the information present in the file and displays it on the screen.

```
// Program to create file with single constructor
 #include<iostream.h>
 #include<fstream.h>
int main()
  {
      ofstream outf(" file1");       //connects ifile1 to outf
      cout<< "Enter item name:";
       char name[30];                //get name from keyboard and write it to file
                                     named file1
       cin>> name;
       outf << name << "\n";
       cout << "Enter item cost:";
       float cost;
 cin>> cost;     // input the value of cost
 outf << cost << "\n";   //write to file file1
 outf.close();             //disconnect file named file1 from outf
 ifstream inf("file1");    // connect file named file1 to inf
 inf >> name;              // read name from file1
inf >> cost;               // read cost from file named file1
cout << "\n";
cout << "Item name:" << name << "\n";
cout << Item cost :" << cost << "\n";
inf.close();               //disconnect file named file1 from inf
return 0;
}
```

The output of the above program will be:

```
Enter item name: Book
Enter item cost: 200
Item name: Book
Item cost: 200
```

2. Opening a file using open() function

As mentioned earlier, the open() function can be used to open multiple files that use the same stream object. For example, sometimes we want to process a set of files sequentially. Then we create a single filestream object, and use it to open each file in turn. This can be done as follows: file-stream-class stream-object;

```
stream-object.open (“filename”);
Example:
ofstream outfile;                // create stream (for output)
outfile.opne(“data1”); //connect stream to data1
 ………..
…………..
outfile.close();          //disconnect stream from data1
outfile.open(“data2”); //connect stream to data2
…………..
………....
outfile.close();          //disconnect stream from data2
…………...
…………..
```

The above program opens two files in sequence for writing data. It is to be noted that the first file is closed before opening the second file. This is required to be done because stream can be connected to only one file at a time.

The files can also be opened in the following way:

The program below illustrates the way to work with multiple files.

```
            //Creating files with open() function
            #include<iostream.h>
    #include<fstream.h>
            int main()
            {
              ofstream fout;                //create output stream object
               fout.open(“country”);                //connect “country” to it
               fout<< “India \n”;
               fout << “USA\n”;
               fout << “UK\n”;
```

```
                    fout.close();           //disconnect “country”
                    fout.open(“capital”);            //connect capital
          fout <<“Delhi\n”;
         fout << “Washington\n”;
        fout << “London\n”;
        fout.close();              //disconnect “capital”
                //Reading the files
          cons tint N=80;                    //size of line
                char line[N];
          ifstream fin;                      //create input strea,
                fin.open(“country”);                //connect “country” to it
                cout << “contents of country file\n”;
                while(fin)                          //check end-of-file
                  {
                    fin.close(line,N);    //read a line
                          cout << line;             //display it
                  }
                fin.close();
                fin.open(“capital”);                //connects “capital”
                cout << “\n contents of capital file \n”;
                while (fin)
                  {
                          fin.getline(line, N);
                          cout << lie;
                  }
                fin.close();
                return 0;
        }
```

The output of the above program will be:

```
Contents of country file
 India
 USA
  Uk
Contents of capital file
 Delhi
   Washington
   London
```

Sometimes it is required to use two or more files in the program simultaneously. For an example, we may require to merge two files into a third sorted file. This means, both the sorted files should be kept open for reading and third one should be kept open for writing. In such cases, it is needed to create two separate input streams for handling the two input files and one output stream for handling the output file.

We have used ifstream and ofstream constructors and the open() function to create new files as well as to open the existing files. You can note that in both the methods only one argument has been used, and that is the name of the file. However, these functions can take two arguments, the second one being for specifying the file mode. The syntax of open() function that accepts two arguments is as follows:

stream-object.open(“filename”, mode);

The second argument, i.e. mode, describes the purpose for which the file is opened. Then the question may arise in your mind as to how we opened the files without providing the second argument in the previous examples.

The prototype of these class member functions contains default values for the second argument, and therefore, they use the default in the absence of the actual values. The following are the default values:

ios::in for ifstream meaning open for reading only.

ios::out for ofstream meaning open for writing only.

The file mode parameter can take one (or more) of such constants defined in the class ios. The table 11.1 shows the file mode parameters.

Table 11.1: File mode parameters

ios::in	Open file to read
ios::out	Open file to write
ios::app	All the date you write, is put at the end of the file. It calls ios::out
ios::ate	All the date you write, is put at the end of the file. It does not call ios::out
ios::trunk	Deletes all previous content in the file. (empties the file)
ios::nocreate	If the file does not exists, opening it with the open() function gets impossible.
ios::noreplace	If the file exists, trying to open it with the open() function, returns an error.
ios::binary	Opens the file in binary mode.

In fact, all these values are int constants from an enumerated type. But, for making your life easier, you can use them as you see them in the table. Here is an example for the method to use the open modes:

```
#include <fstream.h>
void main() {
   ofstream SaveFile("file1.txt", ios::ate);
    SaveFile << "That's new!\n";
    SaveFile.close();
}
```

As you see in the table, using ios::ate writes at the end of the file. If it wasn't used, the file would have been overwritten. So, if file1.txt has this text:

Hi! This is test from www.cpp-home.com!

running it will add "That's new!" to it; so it will look this way:

Hi! This is test from www.cpp-home.com! That's new!

If you want to set more than one open mode, just use the OR operator (**|**) this way:

ios::ate | ios::binary

Using different open modes helps make file handling an easy job. Having

the liberty to choose a combination of these, in the same way, comes very handy in using streams effectively, and to the requirements of the project.

Moving on to something more intriguing and important, we can create a file stream handle, which you can use to read/write file at the same time. Here is how it works:

fstream File("cpp-home.txt", ios::in | ios::out);

In fact, that is only the declaration. The code line above creates a file stream handle, named *File*. As you know, this is an object from class fstream. When using fstream, you should specify ios::in and ios::out as open modes. This way, you can read from the file, and write in it, at the same time, without creating new file handles. Well, of course, you can only read or write. Here is the code example:

```
#include <fstream.h>
void main()
```

```
{
  fstream File("test.txt", ios::in | ios::out);
   File << "Hi!";          //put “Hi!” in the file
   static char str[10];   //when using static, the array is automatically
                          //initialized, and very cell NULLed
   File.seekg(ios::beg); //get back to the beginning of the file
                          //this function is explained a bit later
   File >> str;
   cout << str << endl;
   File.close();
}
```

Let us now understand the above program:

fstream File(“test.txt”, ios::in | ios::out);
This line, creates an object from class fstream. At the time of execution, the program opens the file test.txt in read/write mode. This means that you can read from the file and put data into it at the same time.

File << “Hi!”;

This statement writes Hi! in the file named test.txt

static char str[10];

This makes a char array with 10 cells. The word *static* initializes the array when at the time of creation.

File.seekg(ios::beg);

To understand this statement, consider the following statement:

```
while(!OpenFile.eof())  // here OpenFile is a stream-object
{
   OpenFile.get(ch);
   cout << ch;
}
```

This is ‘a while loop’ that will loop until you reach the end of the file. But how does the loop know if the end of the file is reached? The answer is; when you read the file, there is something like an inside-pointer (current reading/writing position) that shows where you are, with the reading (and writing, too). Every time you call OpenFile.get(ch), it returns the current character to the *ch* variable, and moves the inside-pointer one character after that, so that the next time this function is called, it returns the next character. And this repeats, until you reach the end of the file.

Going back to the code line, the function seekg() will put the inside-pointer to a specific place (specified by you). One can use:

- ios::beg – to put it in the beginning of the file
- ios::end – to put it at the end of the file

Or you can also set the number of characters to go back or after. For example, if you want to go 5 characters back, you should write:

File.seekg(-5);

If you want to go 40 characters after, just write:

File.seekg(40);
It is imperative to mention that the seekg() function is overloaded, and it can take two parameters, too. The other version is this one:

File.seekg(-5, ios::end);

In this example, you will be able to read the last 4 characters of the text, because:

- You go to the end (ios::end)
- You go 5 characters before the end (-5)

Why do you read 4 but not 5 characters? One character is lost, because the last thing in the file is neither a character nor white space. It is just position (i.e., end of file).

Why was this function used in the program above? After putting “Hi!” in the file, the inside-pointer was set after it, i.e., at the end of the file. And as we want to read the file, there is nothing that can be read at the end. Hence, we have to put the inside-pointer at the beginning. And that is exactly what this function does.

File >> str;

This line looks similar to cin >>. In fact, it has much to do with it. This line reads a word from the file, and puts it into the specified array. For example, if the file has this text:

Hi! Do you know me?

Using File >> str, will put just “Hi!” to the *str* array. And, as what we put in the file was “Hi!” we don’t need to use a while loop, that takes more time to code. That’s why this technique was used. By the way, in the while loop for reading that has been used so far, the program reads the file, character by character. But you can read it word by word, this way:

```
char str[30]; //the word can’t be more than 30 characters long
while(!OpenFile.eof())
{
   OpenFile >> str;
   cout << str;
}
```

You can also read it line by line, this way:

```
char line[100]; //a whole line will be stored here
while(!OpenFile.eof())
 {
   OpenFile.getline(line,100); //where 100 is the size of the array
   cout << line << endl;
}
```

It is recommended that you use the line-by-line one, or the first technique that was mentioned, i.e., the one which reads char-by-char. The one that reads word-by-word is not a good option since it will not read the new line. So if you have a new line in the file, it will not display it as a new line, but will append the text to the existing one. But using getline() or get() will show you the file just as it is.

11.2.2 Checking for Failure with File Commands

Now, we will see how to check whether the file opening was successful or not. In fact, there are a few good ways to check it thus, and we will learn some of them. Notice that where there is X, it can be either “o”, or “i”, or nothing (it will then be fstream object).

Example 1: The most usual way

```
Xfstream File(“cpp-home.txt”);
if (!File)
{
   cout << “Error opening the file! Aborting…\n”;
   exit(1);
}
```

Example 2: If the file is created, return an error

```
ofstream File("unexisting.txt", ios::nocreate);
if(!File)
{
   cout << “Error opening the file! Aborting…\n”;
   exit(1);
}
```

Example 3: Using the fail() function

```
ofstream File("filer.txt", ios::nocreate);
if(File.fail())
{
   cout << “Error opening the file! Aborting…\n”;
   exit(1);
}
```

You can see a function fail() in the above example. A non-zero value is retuned by this function if any I/O occurs. There is an interesting fact that needs to be mentioned here. Say, you have created a file stream, but you have not opened a file, as in the following way:

```
ifstream File; //it could also be ofstream
```

This way, we have a handle, but we still have not opened the file. If you want to open it later, it can be done with the open() function (which has already been covered in this chapter). But if anywhere in your program, you need to know whether currently there is an opened file, you can check it with the function is_open(). It retunrs 0 (false) if the file is not opened, and 1 (true) if there is an opened file. For example:

```
ofstream File1;
File1.open("file1.txt");
cout << File1.is_open() << endl;
```

The code above, will return 1 as we open a file (on line 2). But the code below will return 0, because we do not open a file, but just create a file stream handle:

```
ofstream File1;
cout << File1.is_open() << endl;
```

Self Assessment Questions

1. There are two ways to open a file. These are__________ and ___________.
2. The function of _________ is to put at the end of the file all the date we write.
3. The function seekg() will put the inside-pointer to a specific place [True/False].
4. The __________ function can be used to open multiple files that use the same stream object.

11.3 Checking the I/O Status – Flags

The Input/Output system in C++ holds information about the result of every I/O operation. The current status is kept in an object from type *iostate*, which is an enumerated type (just like *open_mode*) defined by ios that has the values described in table 11.2:

Table 11.2: Error-Status flags

Goodbit	No errors(no flags set, value=0)
Eofbit	End of file has been reached
Failbit	Non-fatal I/O error
Badbit	Fatal I/O error

There are two ways to receive information about the I/O status. One of them is by calling the function rdstate() which is a member of ios. It returns the current status of the error-flags (the above mentioned). For example, the *goodbit* is returned by the function rdstate() if no errors are encountered.

The other way to check the I/O status is by using any of the following functions:

```
bool bad();
bool eof();      //Read until the end of the file has been reached
bool fail(); /   /Check if the file opening was successful
bool good();
```

The function bad() returns true, if the badbit is set. The fail() function returns true if the failbit is set. The good() function returns true if there are no errors (the goodbit bit is set). And the eof() function returns true if the end of the file has been reached (the eofbit is set.).

Once an error has occurred, it might need to be cleared before your program continues. To do this, you have to use the ios member function clear() whose prototype is as follows:

void clear(iostate flags=ios::goodbit);

If flags is goodbit (as it is by default) as error flags are cleared. Otherwise, you can set flags to the desirable settings.

Some examples to show the use of flags are given below:

Example 1: Simple status check

```
//Replace FileStream with the name of the file stream handle
if (FileStream.rdstate() == ios::eofbit)
   cout << "End of file!\n";
if (FileStream.rdstate() == ios::badbit)
   cout << "Fatal I/O error!\n";
if (FileStream.rdstate() == ios::failbit)
   cout << "Non-fatal I/O error!\n";
if (FileStream.rdstate() == ios::goodbit)
   cout << "No errors!\n";
```

Example 2: The clear() function

```
#include <fstream.h>
void main() {
   ofstream File1("file2.txt"); //create file2.txt
   File1.close();
   //this will return error, because ios::noreplace is used
   //open_mode, which returns error if the file already exists.
   ofstream Test("file2.txt", ios::noreplace);
   //The error that the last line returned is ios::failbit
   if (Test.rdstate() == ios::failbit)
      cout << "Error...!\n";
   //set the current status to ios::goodbit
   Test.clear(ios::goodbit);
   //check if it was set correctly
   if (Test.rdstate() == ios::goodbit)
      cout << "Fine!\n";
   Test.clear(ios::eofbit); //set it to ios::eofbit. Useless.
   //and check again if it is this flag indeed
   if (Test.rdstate() == ios::eofbit)
      cout << "EOF!\n";
   Test.close();
}
```

Self Assessment Questions

5. The function bad() returns true, if the badbit flag is set. (True/False).
6. ______________ function returns the current status of the error-flags.

11.4 Dealing with Binary Files

Although it is best to work with formatted files, sometimes we need to work with unformatted files, i.e. binary files, too. They have the same look as your program, and they are much different from what comes after using the << and >> operators. The functions that give you the possibility to write/read unformatted files are get() and put(). To read a byte, you can use get(), and to write a byte, put().

get() and put() both take one parameter, a char variable or character. If you want to read/write whole blocks of data, then you can use the read() and write() functions. Their prototypes are:

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

For the read() function, buf should be an array of chars, where the read block of data will be put. For the write() function, buf is an array of chars, where the data you want to save in the file is. For both the functions, num is a number that defines the amount of data (in symbols) to be read/written.

If you reach the end of the file, before you have read “num” symbols, you can see how many symbols were read by calling the function gcount(). This function returns the number of read symbols for the last unformatted input operation. And, before going to the code examples, we must take note that if you want to open a file for binary read/write, you should pass ios::binary in an open mode.

Now, let us see some code examples so that you can see how stuff works.

Example 1: Using get() and put()

```
#include <fstream.h>
void main() {
   fstream File("test_file.txt",ios::out | ios::in | ios::binary);
    char ch;
    ch='o';
    File.put(ch); //put the content of ch to the file
    File.seekg(ios::beg); //go to the beginning of the file
    File.get(ch); //read one character
    cout << ch << endl; //display it
    File.close();
}
```

Example 2: Using read() and write()

```
#include <fstream.h>
#include <string.h>
void main() {
    fstream File("test_file.txt",ios::out | ios::in | ios::binary);
    char arr[13];
    strcpy(arr,"Hello World!"); //put Hello World! into the array
    File.write(arr,5); //put the first 5 symbols into the file- "Hello"
    File.seekg(ios::beg); //go to the beginning of the file
    static char read_array[10]; //I will put the read data, here
    File.read(read_array,3); //read the first 3 symbols- "Hel"
    cout << read_array << endl; //display them
    File.close();
}
```

Self Assessment Questions

7. The functions that allow you to write/read unformatted files are get() and put().
8. To read a block of data from the file _______________ function is used.

11.5 Some useful functions

- **tellg() –**function is used to get the current position of the input pointer in the file. This function returns an integer type value.

  ```
  #include <fstream.h>
  void main()
  {
  ```

```
//if we have "Hello" in test_file.txt
ifstream File("test_file.txt");
char arr[10];
File.read(arr,10);
//this should return 5, as Hello is 5 characters long
cout << File.tellg() << endl;
File.close();
}
```

- **tellp() –** this is the same as *tellg()* but is used when we write in a file. This function is used to get the current position of the output pointer in the file.
- **seekp() –** You have already studied seekg() function which is used to move the input pointer to a specified location. Similarly, seekp() function is used to move the output pointer to a specified location. seekp() function can be used in the same way as seekg() function.
- **ignore() –** Used when reading a file. If you want to ignore certain number of characters, this function can be used to extract and discard them from the input stream. In fact, you can use seekg() instead, but the ignore() function has one advantage: you can specify a delimiter rule, where the ignore() function will stop. The syntax of ignore() is as follows:

  ```
  stream_obj.ignore(int count, delimiter);
  ```

  Here the ignore() function will continue to extract and discard the characters from the input stream until the count characters are discarded or the character specified by the delimiter is encountered.

  For example:

  ```
  ignore(5,h);
  ```

  Here ignore() function contains two arguments i.e. an integer value 5 and a character value h. So the ignore function continues to ignore characters until either 5 characters have been discarded or the character h is found.
- **getline() –**The getline() function is used for the read operation. This function can be used to read the contents of the file line by line from an

input stream, but it can be set to stop reading if it met a certain symbol. Here is how you should pass the parameters to it:

getline(array, array_size, delim);

And here is a code example:

```
#include <fstream.h>
void main() {
//if we have "Hello World" in test_file.txt
Ifstream File("test_file.txt");
static char arr[10];
/*read, until one of these happens:
  1) You have read 10
  2) You met the letter "o"
  3) There is a new line
*/
File.getline(arr, 10, 'o');
cout << arr << endl; //it should display "Hell"
File.close();
}
```

- **peek()** – This function will return the next character from an input file stream, but unlike get() function, it won't move the inside-pointer. get(), for example, returns the next character in the stream, and after that, it moves the inside-pointer, so that the next time you call the get() function, it will return the next character, but not the same one. Using peek() will return a character, but it won't move the cursor. So, if you call the peek() function twice in succession, it will return the same character. The syntax to use the peek() function is as follows:

  stream_obj.peek();

  In the above syntax peek() function will return the next character or eof if the end of the file is reached.

  Consider the following code example:

```
#include <fstream.h>
void main()
{
//if we have "Hello World" in test_file.txt
ifstream File("test_file.txt");
char ch;
File.get(ch);
cout << ch << endl; //should display "H"
cout << char(File.peek()) << endl; //should display "e"
cout << char(File.peek()) << endl; //should display "e" again
File.get(ch);
cout << ch << endl; //should display "e" again
File.close();
}
```

The *peek()* function actually returns the ASCII code of the char, but not the char itself. So, if you want to see the character itself, you have to call it the way shown above.

- **remove()**

  The files can be removed by calling the remove function which has the following specification:

```
#include<stdio.h>
int remove(const char *filename);
```

The remove() function deletes a file whose name is the string pointed to by fname. If successful, it returns zero, or else, it returns a non-zero value. If fname is open, i.e. if it is associated with a stream and this association has not been broken, then the behavior of remove is not defined.

Let us see the following example:

```
/* remove example: remove myfile.txt */
#include <stdio.h>
int main ()
{
if( remove( "myfile.txt" ) != 0 )
perror( "Error deleting file" );
else
puts( "File successfully deleted" );
return 0;
}
```

If the file myfile.txt exists before the execution and the program has write access to it, the file would be deleted and the following message would be displayed to stdout:

```
File successfully deleted
```

Otherwise, a message similar to this would be written to stderr:

```
Error deleting file: No such file or directory
```

- **putback() –** The putback() function is a member of istream.

  It attempts to decrease the current location in the stream by one character, making the last character extracted from the stream once again available to be extracted by input operations. Here is a code example:

```
#include <fstream.h>
void main() {
//test_file.txt should have this text- "Hello World"
ifstream File("test_file.txt");
char ch;
File.get(ch);
cout << ch << endl; //it will display "H"
File.putback(ch);
cout << ch << endl; //it will again display "H"
File.get(ch);
cout << ch << endl; //it will display "H" again
File.close();
}
```

- **flush() –**Whenever you perform a write operation on a file, the contents are not written directly, but are saved in a buffer (storage space).

  And when the buffer gets filled, then the data is put in the real file (on your disk). Then the buffer is emptied, and so on. But if you want to save the data from the buffer, even if the buffer is still not full, use the flush() function. Just call it this way- FileHandle.flush(). Consequently, the data from the buffer will be put in the physical file, and the buffer will be emptied.

  The syntax of the flush() function is as follows:

```
stream_obj.flush();
```

Self Assessment Questions

9. _________ function is used to get the current position of the input pointer in the file.
10. _________ function is used to ignore certain amount of characters from the input stream.
11. _________function decreases the current location in the stream by one character, making the last character extracted from the stream once again available to be extracted by input operations.
12. The peek() function will return the next character from an input file stream, but like get() function, it will move the inside-pointer. (True/False)

11.6 Summary

- This unit focusses on how to manage the file to operate using I/O streams. This chapter has discussed comprehensively the C++ streams. Various examples provided throughout this chapter illustrate the use of these streams.
- For opening a file you have to create a file stream, and it is to be linked to a filename. You can define a filename using ifstream, ofstream and fstream classes. These classes are contained in the header file fstream.
- There are two ways to open a file. They are: using the constructor function of the class, and using the member function open() of the class.
- The open() function is used to open multiple files that use the same stream object.
- There are two ways to receive information about the I/O status. One of them is by calling the function rdstate(). And the other way to check the I/O status is by using any of the following functions: bool bad(), bool eof(), bool fail() and bool good().
- The functions that give you the possibility to write/read unformatted files are get() and put(). To read a byte, you can use get(), and to write a byte, use put().
- Some useful functions in c++ are: tellg(), tellp(), seekp(), ignore()
- getline(), peek(), remove(), putback() and flush().

11.7 Terminal Questions

1. Explain the types of methods to open a file.
2. Discuss the methods to check whether the file opening was successful or not.
3. Discuss 'on file I/O flags'.
4. Elaborate the concept of binary files.
5. List and discuss some important functions of file operations.

11. 8 Answers

Self-Assessment Questions

1. using constructor function of the class, using member function open() of the class.
2. ios::app
3. True
4. open()
5. True
6. rdstate()
7. get() and put()
8. read()
9. tellg()
10. ignore()
11. putback()
12. Flase

Terminal Questions

1. There are two ways to open a file. These are: using the constructor function of the class, and using the member function open() of the class. For more details refer section 11.2.1.
2. The most usual way is:

   ```
   Xfstream File("cpp-home.txt");
   if (!File)
   {
   cout << "Error opening the file! Aborting…\n";
   exit(1);
   ```

   For more details refer section 11.2.2.

3. The Input/Output system in C++ holds information about the result of every I/O operation. The current status is kept in an object from type *iostate*, which is an enumerated type (just like *open_mode*) defined by ios. For more details refer section 11.3.
4. Although it is best to work with formatted files, sometimes we need to work with unformatted files, i.e. binary files, too. They have the same look as your program, but it is much different from what comes after using the << and >> operators. For more details refer section 11.4.
5. tellg(), tellp() etc. are some of the functions that reflect the position of the current pointer in flags. For more details refer section 11.5.