
Chapter Table of Contents

Chapter 5.2

Streams and Files

Aim.....	269
Instructional Objectives.....	269
Learning Outcomes.....	269
5.2.1 Introduction.....	270
5.2.2 Console I/O operator.....	270
(i) C++ Stream and C++ Stream Class	271
(ii) Manipulators.....	274
(iii) Unformatted I/O Operators	276
(iv) Formatted I/O Operators	281
Self-assessment Questions.....	282
5.2.3 Files	283
(i) Class for the File Stream Operators	284
(ii) Opening and Closing a File.....	285
(iii) Reading and Writing Objects to Disks.....	289
(iv) Binary versus Character Files.....	291
(v) I/O with Multiple Objects	292
Self-assessment Questions.....	295
5.2.4 File Pointers	295
(i) Specifying the Position	295
(ii) Specifying the Offset.....	296
(iii) Seekg() function	297
(iv) Tellg() Function.....	298
Self-assessment Questions.....	299
5.2.5 Command Line Arguments	300
Self-assessment Questions.....	301
Summary	302
Terminal Questions.....	303
Answer Keys.....	304

Activity.....	304
Bibliography.....	305
e-References	305
External Resources	305
Video Links	305



Aim

To equip the students with the skills to use streams and files in writing C++ programs



Instructional Objectives

After completing this chapter, you should be able to:

- Explain console I/O Operator
- Brief about formatted and unformatted I/O operators
- Explain file handling in C++ with example
- Illustrate reading and writing an object from and to the disk by opening and closing a file.
- Demonstrate programs on `tellg()` function and `seekg()` function
- Elaborate on command line arguments



Learning Outcomes

At the end of this chapter, you are expected to:

- Compare C++ stream and C++ stream class
- Elaborate user defined manipulators and manipulators with one parameter
- Differentiate between formatted and unformatted I/O operators
- Explain how would you determine number of objects in a file
- Explain the concept of input output with multiple object and stream classes
- Identify `tellg()` function and `seekg()` function
- Write a code to print the data of file onto command line

5.2.1 Introduction

Till now we have learnt the usage of the `iostream` standard library. It provides `cin` and `cout` methods for reading from standard input and writing to standard output respectively.

Now we will see how to read from a file and also how to write to a file. In order to learn this we need to know about another standard library of c++ called `fstream` library. This consists of new three set of data types along with the description .

- **ofstream** – This data type represents the output file stream and is used to create files and to write information to files,
- **ifstream** – This data type represents the input file stream and is used to read information from files.
- **Fstream** – This data type represents the file stream generally and has the capabilities of both `ofstream` and `ifstream`. This means it can create files, write information to files and read information from files.

To perform file processing in C++, header files and must be included in your C++ source file.

5.2.2 Console I/O operator

The console I/O operators are those which accept the inputs and output through the user and displays on the console. This is done through the console of the system. The following are the two console I/O operators – `cin` and `cout`.

The following code describes the working of the above console I/O operators:

Program:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main( )
6  {
7      char name[50];
8
9      cout << "Please enter your name: ";
10     cin >> name;
11     cout << "Your name is: " << name << endl;
12
13 }
14
```

Output:

```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Please enter your name: rahul
Your name is: rahul
sh-4.3$
```

(i) C++ Stream and C++ Stream Class

Stream

The I/O system of C++ gives the software engineer an interface which is free of the real gadget being used. This interface is the stream. A stream is an arrangement of bytes. It can be a source from where the input is acquired or can be a destination where the output information can be sent. The source from where the input is given is called as the input stream, while the place where the output is obtained is called the output stream.

The following figure 5.2.1 explains how the I/O system of C++ handles streams for input and output devices.

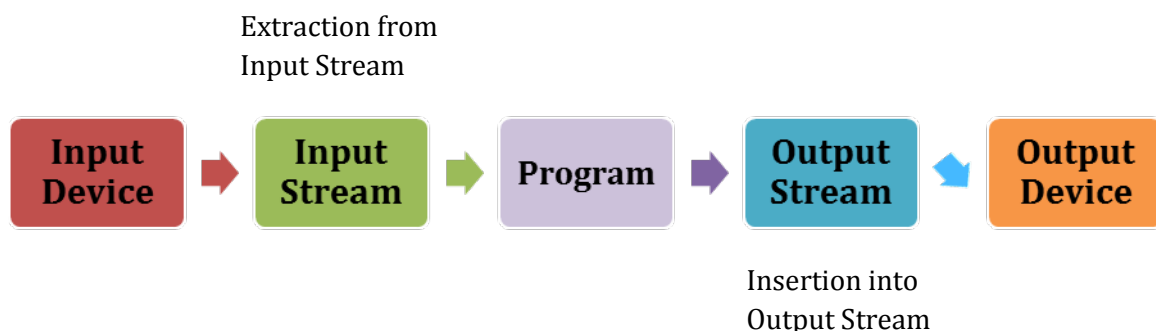


Figure 5.2.1: Stream for Input and Output Device

The information which is in the input gadget can be from any of the input gadgets like console or some other storage gadgets. Similarly the handled information on the output stream can be shown on the screen or can be put away to any storage gadgets. Along these lines a C++ program can deal with the information free of the gadgets utilized. Alongside this, C++ has numerous predefined streams that are naturally opened when a program begins its execution. One of such is cin and cout functions which we utilized as a part of our initial program. As we

are probably aware of the fact that cin and cout corresponds to the input stream and output streams separately.

Stream Class

The I/O system contains an arrangement of chain of importance of the stream classes used to characterize different streams to manage the console and storage files. This arrangement of the classes is called as stream classes. The accompanying diagram demonstrates the chain of command of the stream classes which are utilized for the input and output operations alongside the console unit. The classes utilized here are characterized and pronounced in the header record iostream, which must be incorporated into the program for a smoother correspondence with the console unit.

The following figure 5.2.2 explains the different stream classes for the console input and output.

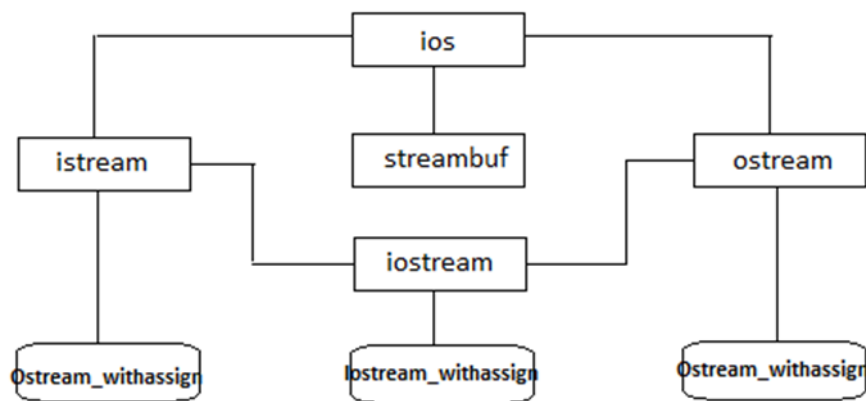


Figure 5.2.2: Stream classes for the Console Input and Output

In the above chart, ios is the base class from where the istream (input stream) and the ostream (yield stream) branch out, which are the base classes for the iostream. So the class ios is announced as the virtual base class so that one and only duplicate of its individuals are acquired by the iostream.

This virtual base class gives the essential backing to the arranged and unformatted I/O operations. Arranged and unformatted I/O operations get the facilities from the class istream. The ostream class gives facilities just to the organized yield through inheritance. The iostream class gives the offices to take care of both input and output streams. The below table portrays the input and output stream classes which are considered above.

Table 5.2.1: Stream Classes for the Console I/O Operations

Class name	Contents
ios (General input/output stream class)	Contains essential features that are used by all other i/o classes
	Likewise contains a pointer to a buffer object (streambuf object)
	Declare constants and functions that are necessary for handling formatted input and output operations
istream (input stream)	Inherits the properties of ios
	Declares input functions such as get() , getline() and read()
	Contains overloaded extraction operator >>
ostream (output stream)	Inherits the properties of ios
	Declares output functions put() and writer()
	Contains overloaded insertion operator <<
iostream (input/output stream)	Acquires the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streambuf	Provides an interface to physical device through buffers
	Acts as base for filebuf class used ios files

The iostream class declares a portion of the objects that are utilized to perform i/o operations as a major aspect of the library. These are known as Stream objects. They are cin, cout, cerr, obstruct and so on.

(ii) Manipulators

Manipulators are group of functions which are given by the header file `iostream`, which can be utilized to control the output formats. The components are like that of `ios` member functions and flags. A portion of the manipulators are more advantageous to utilize superior to their partners in the class `ios`, as two or more manipulators can be utilized as a chain in the explanation given below –

```
cout << manip1 << manip2 << manip3 << item;  
cout << manip1 << item1 << manip2 << item2;
```

The above concatenation will be useful when we need to show the output in several sections of output.

The most frequently utilized manipulators are depicted as a part of the underneath table with their explanations. Keeping in mind the end goal of these manipulators is to incorporate the file `iostream` into the program.

Table 5.2.2: Manipulators and their Meanings

Manipulator	Meaning	Equivalent
<code>setw(int w)</code>	Set the field width to <code>w</code>	<code>width()</code>
<code>setprecision(int d)</code>	Set the floating point precision to <code>d</code>	<code>precision()</code>
<code>setfill(int c)</code>	Set the fill character to <code>c</code>	<code>setfill()</code>
<code>setiosflags(long f)</code>	Set the format flag <code>f</code>	<code>setf()</code>
<code>resetiosflags(long f)</code>	Clear the flag specified by <code>f</code>	<code>unsetf()</code>
<code>endl</code>	Insert new line and flush stream	<code>"\n"</code>

Some of the examples for the manipulators are as shown below:

```
Cout << setw(5) << abcd;
```

The above explanation prints the character "abcd" right justified in a field width of 5 characters. The output can likewise be adjusted for left – justification by changing the above statement given below:

```
Cout << setw(5) << setiosflags(ios::left) << abcd;
```

Manipulators are functions particularly intended to be utilized in conjunction with the inclusion (<<) and extraction (>>) operators on stream objects.

User Defined Manipulators:

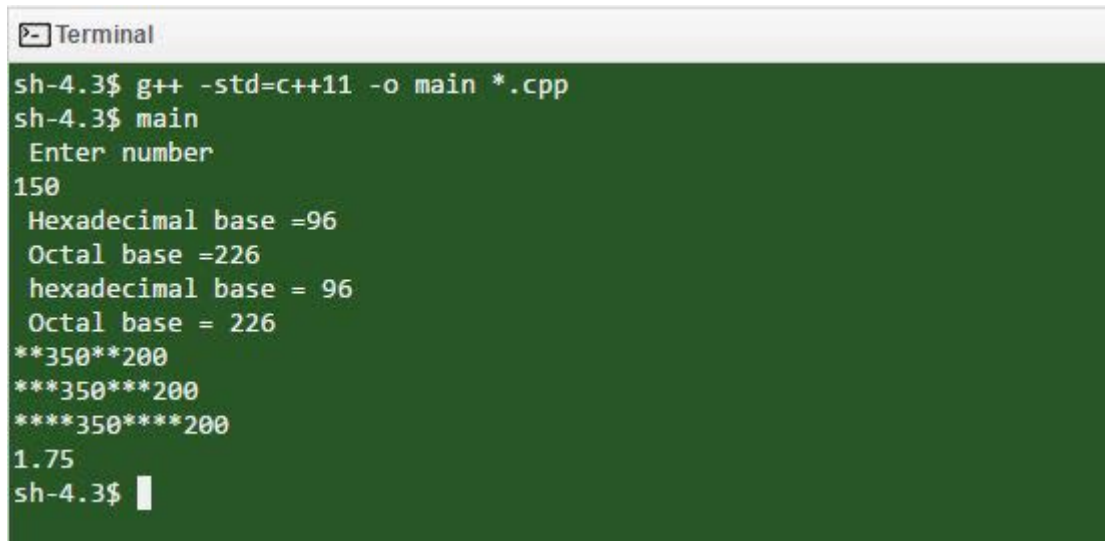
C++ gives a collection of predefined manipulators. The header file iomanip.h consists of these manipulators. Besides, you can outline your own manipulators in C++ to suit for any particular functionality. The user defined manipulators are defined below:

```
Ostream & manipulator (ostream & ostr)
{
    set of statements;
    return ostr;
}
```

Consider the following example:

```
1  #include<iostream>
2  #include<iomanip>
3
4  using namespace std;
5
6  int main ()
7  {
8  int value;
9  float a,b,c;
10 a = 350;
11 b = 200;
12 c = a/b;
13
14 cout << " Enter number" << endl;
15 cin >> value;
16 cout << " Hexadecimal base =" << hex << value << endl;
17 cout << " Octal base =" << oct << value << endl;
18 cout << " hexadecimal base = " << setbase (16);
19 cout << value << endl;
20 cout << " Octal base = " << setbase (8) << value << endl;
21 cout << setfill ('*');
22 cout << setw (5) << a << setw (5) << b << endl;
23 cout << setw (6) << a << setw (6) << b << endl;
24 cout << setw (7) << a << setw (7) << b << endl;
25
26 cout << fixed << setprecision (2) << c << endl;
27
28 }
```

Output:



```
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Enter number
150
Hexadecimal base =96
Octal base =226
hexadecimal base = 96
Octal base = 226
**350**200
***350***200
****350****200
1.75
sh-4.3$
```

Manipulators are specialized stream functions that are utilized to change certain attributes of the input and output. By incorporating `<iomanip.h>` header file, we can use manipulators as a part of your C++ program. The following list provides some of the important manipulators that can be incorporated in our C++ programs.

- | | |
|-----------------|-----------------|
| - endl | - hex, dec, oct |
| - setbase | - setw |
| - setfill | - setprecision |
| - ends | - ws |
| - flush | - setiosflags |
| - resetiosflags | |

endl: The endl is an output manipulator that is used to generate a carriage return. It works like “\n”

Setbase(): The setbase() manipulator is used to convert the base of one numeric value into another base.

Setw (): It is used to specify the minimum number of character positions, a variable will consume on the command prompt.

(iii) Unformatted I/O Operators

Overloaded Operators >> and <<

The objects cin and cout are used for input and output of data of various types. This can be done by overloading the operators << and >>. The operator >> is overloaded in the **istream**

class, similarly >> is overloaded in the **ostream** class. The following is the general syntax for reading data from the keyboard:

```
cin >>var1 >> var2 >> ... >> varN;
```

Here var1, var2... are the valid C++ input variable names have been declared earlier. This stops the execution and waits for the input data from the keyboard. The input data would be:

```
dat1, dat2 ...datN
```

The data which is taken as input can be separated by the white spaces and also should match with the type of the variable which is declared earlier.

The data is read by the operator >> character by character and assigns in the location which is indicated. The reading will be stopped when it encounters a white space or when a character which does not match with the type of the destination.

The general syntax of the cout function is as shown below:

```
cout << var1 << var2 << var3.....;
```

The variables var1, var2, var3 may be a variable or a constant of any basic types. All these are illustrated in the previous examples which we are discussed earlier.

put() and get() functions:

get() and **put()** functions, are the two member functions defined in **istream** and **ostream** classes respectively, which handles the single character input/output operations. The following are the two different types of get() functions – **get(char *)** assigns the input character to its argument and **get(void)** returns the input character and also prototypes to fetch a character which includes white spaces, tabs and newline character.

We must invoke these functions using appropriate object since they are the member functions of the I/O stream classes.

Example:

```
Char c;  
Cin.get(char);    //gets a character from the keyboard and assign it to char  
While(char!= '\n')  
{  
    Cout << char;    //display the character on screen  
    Cin >> get(char); // get another character  
}
```

The above code executes and it reads and displays a line of text. Here the operator >> also acts as the input character to read data, but this will skip the white spaces and newline character.

Suppose if we replace,

Cin >> get(char); with cin >> char; then the while loop does not work properly.

The function get(void) works as follows:

```
. . . . .
Char ch;
Ch=cin.get();           //cin.get(char") is replaced
. . . . .
. . . . .
```

In the above example, the value returned by the function **get()** is assigned to ch.

The function **put()** is the member of the member of the ostream class. This can be used to output a line of text, character by character. Consider the following example:

```
Cout.put ( 'p' );
```

Displays the character p and

```
Cout.put( char );
```

Displays the value of the variable char.

Similarly we can also use number as an argument in the put() as follows:

```
Cout.put(65);
```

Which displays the character A, by converting the int value 65 to a char value and displays the ASCII equivalent of the 65.

Consider the following program which illustrate the working of the above said functions –

```
#include < iostream
Using namespace std;
Int main()
{
    Int cnt = 0;
    Char ch;
    Cout << "input text \n";
    Cin.get( ch );
```

```
        While(ch!='\n')
        {
            Cout.put(ch);
            Cnt++;
            Cin.get(ch0);
        }
        Cout << "Number of Characters are : " << cnt << "\n";
    Return 0;
}
```

Output as follows:

```
Input text
Welcome to C++
Output displayed:
    Welcome to C++
    Number of Characters are : 14
```

getline() and write() Functions:

Inorder to read and display a line of texts more efficiently we use line – oriented O/O funtions such as **getline()** and **write()**. The **getline()** reads the whole line of text ends with a newline character. This is invoked by the object in **cin** as follows:

```
Cin.getline(line,size);
```

The above function call invoke the function **getline()** which is used to read character input into the variabe line. This stops reading as soon as it encounters a newline character. The newline character is also read but not stored, instead it is replace by a nill character.

Consider the following example:

```
Char str[20];
Cin.getline( str,20)
```

Assume that we have read the input as follows:

```
Files and Streams <press RETURN>
```

This above set of characters is read completely without giving any error.

Consider the following:

```
Object Oriented Programming using C++ <press RETURN>
```

In this case the, the set of characters read are as follows:

‘Object Oriented pro ‘

Here only 20 character are read from the text because the size is limited to 20. Consider the following example which illustrates the usage of >> and **getline()** functions:

Program:

```
1  #include<iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int size = 30;
8      char cty[30];
9      cout << "Enter the name of the city: " << endl;
10     cin >> cty;
11     cout << "City Name is : "<< cty << endl;
12     cout << "Enter the city name again:"<< endl;
13     cin.getline(cty, size);
14     cout << "Now the city name is: "<< cty << endl;
15     cout << "Enter the another city name:" << endl;
16     cin.getline(cty, size);
17     cout << "The new city name is: "<< cty << endl;
18     return 0;
19 }
```

The output of the above code would be as follows:

Execution 1

```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Enter the name of the city:
mumbai
City Name is : mumbai
Enter the city name again:
Now the city name is:
Enter the another city name:
delhi
The new city name is: delhi
sh-4.3$
sh-4.3$
```

Execution 2:

```
Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main
Enter the name of the city:
pune
City Name is : pune
Enter the city name again:
Now the city name is:
Enter the another city name:
mysore
The new city name is: mysore
sh-4.3$
```

Left to the students for discussion

Discuss a program to implement the **write()** function.

(iv) Formatted I/O Operators

C++ has a number of features which are utilized for organizing the output. Some of the features are:

- Ios functions and flags.
- Manipulators.
- User – defined output functions.

The ios class contains many member functions that aids us to arrange the output multiple ways. Some of them are given below in the table:

Table 5.2.3: IOS Format Functions

Function	Task
width()	To demonstrate the required data size for showing an output value
precision()	To demonstrate the number of digits to be showed after the decimal place of a float value
fill()	To demonstrate a character that is used to fill the unused part of a data field
setf()	To demonstrate format flags that controls the form of output display (such as left-justification and right-justification)
unsetf()	To clear the flags indicated

Manipulators are the special functions which can be incorporated into the Input/Output statements to adjust the arrangement parameters of a stream. The accompanying table gives a portion of the vital manipulators which are mostly utilized.

Table 5.2.4: Manipulators

Manipulators	Equivalent ios function
Setw()	Width()
Setprecision()	Precision()
Setfill()	Fill()
Setiosflags()	Setf()
Resetiosflags()	Unsetf()



Self-assessment Questions

- 1) The base class for most stream classes is the _____ class.
- 2) 2) We can output text to an object of class ofstream using the insertion operator << because,
 - a) The ofstream class is a stream.
 - b) The insertion operator works with all classes.
 - c) We are actually outputting to cout.
 - d) The insertion operator is overloaded in ofstream
- 3) Name three stream classes commonly used for disk I/O.

5.2.3 Files

In our real life, the problem is to store the large volume of data and in such a situation we use some secondary storage devices such as, floppy drives or the hard disks to store the large volumes of data. The data is organised using the concept of **Files**. The definition of the file is – **“Collection of the related data stored in a particular area on the disk”**.

Generally, a program may involves the operations like,

- Data transfer between the console unit and the program
- Data transfer between the program and the disks.

Which is illustrated in the following diagram:

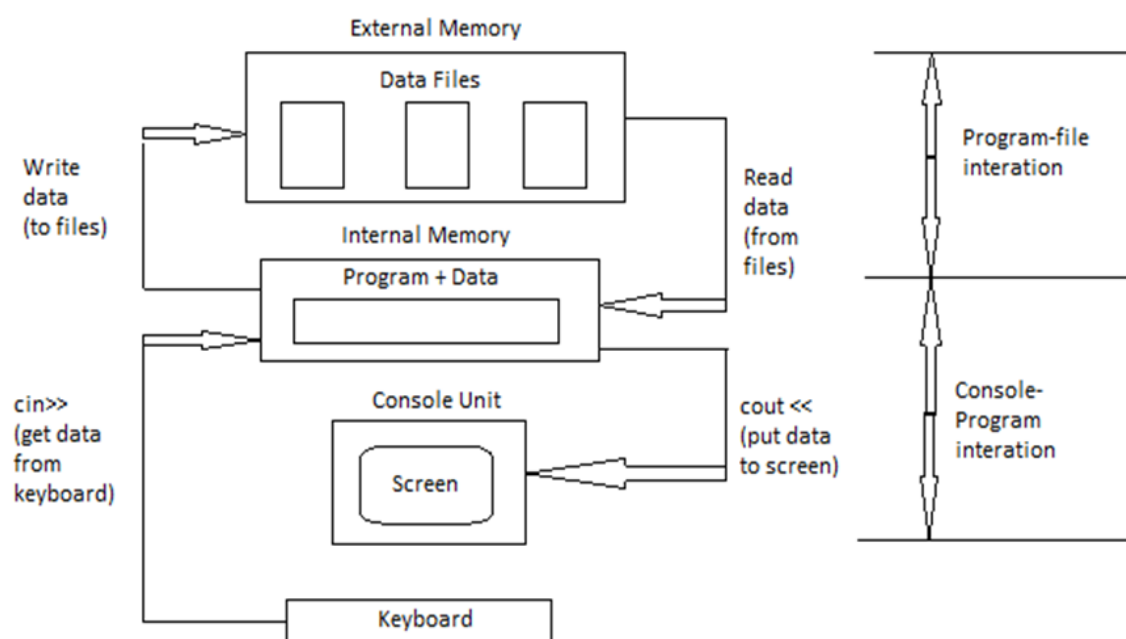


Figure 5.2.3: Console Program File Interaction

So far we have learnt the techniques for handling data between the console and the program. Here in this, we'll study how to handle (storing and retrieving) data between the program and a file.

The following diagram illustrates the file input and output streams.

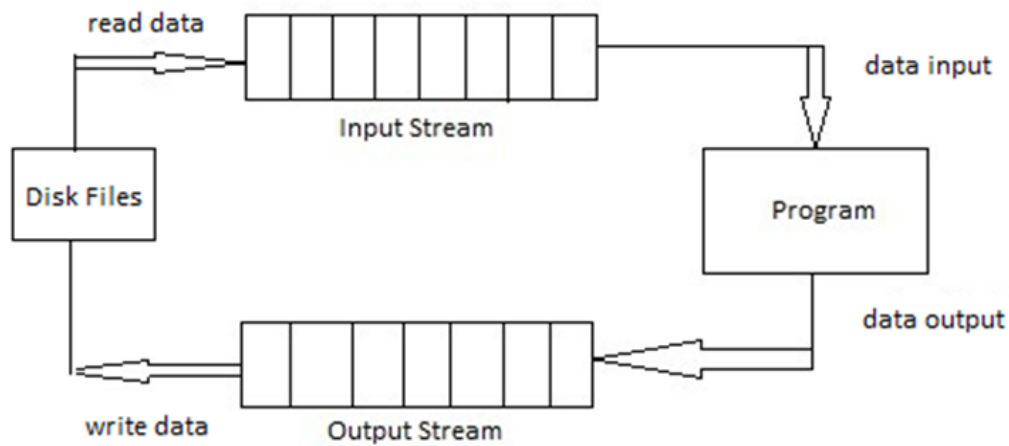


Figure 5.2.4: File Input and Output Stream

The input operation involves in creation of an input stream and links it with the program. Similarly, the output operation involves in the creation of the new links which is used establish an output stream with the necessary links with the program and the respective output file.

(i) Class for the File Stream Operators

The C++ Iostream consists of a set of classes which defines the file handling methods. Those are **ifstream**, **ofstream** and **fstream**. These are the classes which are derived from the base class **fstreambaseclass** and from the corresponding **iostream** class.

The following diagram illustrates the file streams:

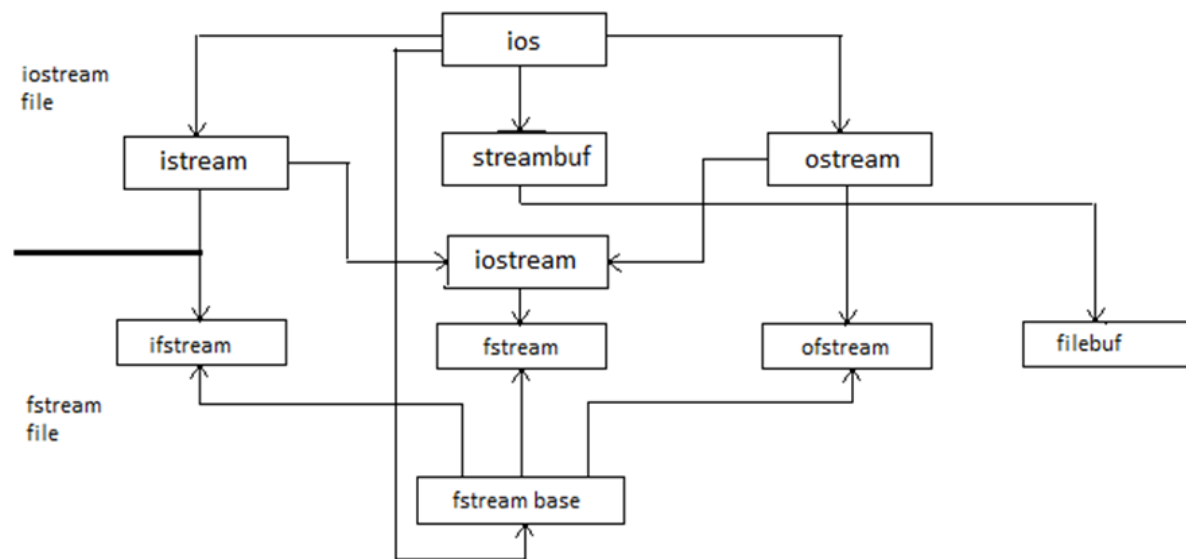


Figure 5.2.5: Stream Classes for File Operations (Contained in Fstream)

(ii) Opening and Closing a File

When we are using a disk file, the following things about the file should be decided before using the file:

- Suitable name for a file
- Data type and Structure
- Purpose of the file
- Opening method of the file

A file can be opened in two methods:

- Using the constructor function of the class
- Using the member function `open()` of the class.

Using the constructor method:

As we know that the constructor is used to initialize object when it is been created. Here the filename is used to initialize the file stream object.

The process involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class, which means the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
2. Initialize the file object with the desired function.

For example, consider the following:

```
Ostream Outfile ("Results");           //output only
```

The statement create's **outfile** as an **ostream** object which manages the output stream. The object created may be any of the valid C++ names such as `o_file`, `my_file` or `fout`. The above statement also opens the file results and attaches it to the output stream **outfile**.

The above is illustrated in the following diagram:

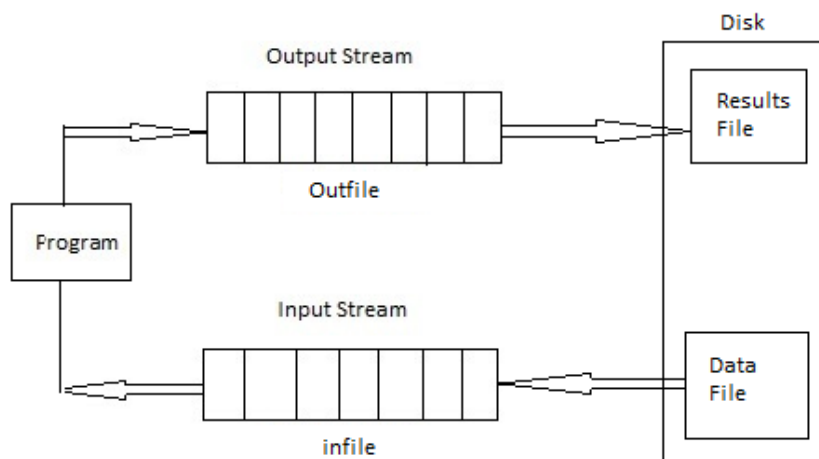


Figure 5.2.6: Two File Streams Working on Separate filesvoid main()

In the same way, the function **infile** works as follows:

```
istream infile ( "input" );
```

Some of the examples are as follows:

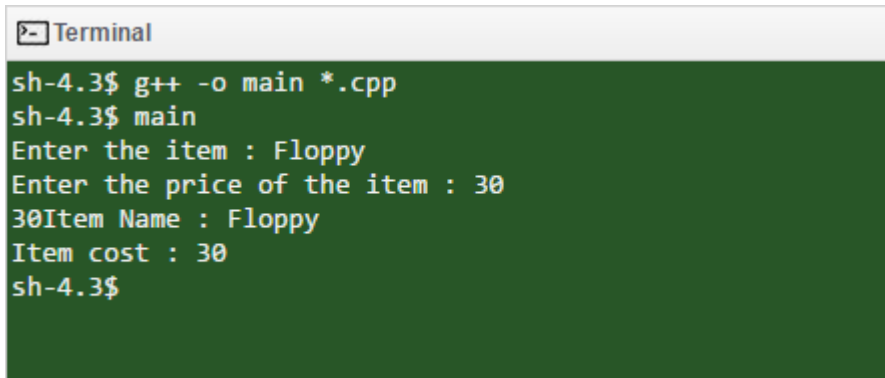
```
outfile << "Result";  
outfile << "average";  
outfile <<"sum";  
infile >>"string ";
```

The following example shows the working of the constructors for writing and reading the data.

```
// Creating files with constructor functions
```

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main()
5  {
6      ofstream outfile ( "ITEM" );
7      cout << "Enter the item : ";
8      char itm[20];
9      cin >> itm;
10     outfile << itm << "\n";
11     cout << "Enter the price of the item : ";
12     float amt;
13     cin >> amt;
14     cout << amt;
15     outfile << amt << "\n";
16     outfile.close();
17     ifstream infile( "ITEM" );
18     infile >> itm;
19     infile >> amt;
20     cout << "Item Name : " << itm << "\n";
21     cout << "Item cost : " << amt << "\n";
22     infile.close();
23     return 0;
24 }
```

When the above code is executed, the following output will be produced:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Enter the item : Floppy
Enter the price of the item : 30
30Item Name : Floppy
Item cost : 30
sh-4.3$
```

Opening files using open():

As we that, the function open() can be used to open multiple files as well, which uses the same stream object. Suppose if we want to process a set of files sequentially, then we should create a single stream object and use it to open each of the file.

This is illustrated as follows:

```
file-stream-class steam_obj;  
stream-object.open ( " file_name " );
```

For example,

```
Ofstream.of_file;           //create stream ( for output )  
Ot_file.open( " DATA_1 " );           //connect stream to DATA_1  
.  
.  
.  
.  
.  
Ot_file.close();           //disconnect stream from DATA_1  
Ot_file.open( " DATA_2 " );           //connect stream to DATA_2  
.  
.  
.  
.  
.  
Ot_file.close();           //disconnect stream to DATA_2
```

In the above segment, two files are opened in sequence for writing the data. Here we should note that the first file should be closed before opening the second file. This has to be done because only one file can be connected through stream.

Similarly the following code illustrates the working with multiple files:

//creating files with open() function

```
1  #include <iostream>  
2  #include <fstream>  
3  using namespace std;  
4  
5  int main()  
6  {  
7      ofstream outfile;  
8      outfile.open( " Country " );  
9      outfile << " United States of America \n";  
10     outfile << " United Kingdom \n";  
11     outfile << " South Korea \n ";  
12     outfile.close();  
13     outfile.open( " Capital " );  
14     outfile << " Washington \n";  
15     outfile << " London \n";  
16     outfile << "Seoul \n ";  
17     outfile.close();  
18     //Reading the files  
19     const int N=80;  
20     char line[N];  
21  
22     ifstream infile;  
23     infile.open ( " Country " );  
24     cout << "Contents of Country File - \n";  
25     while(infile)
```

```

26 {
27     infile.getline(line, N);
28     cout << line;
29 }
30 infile.close();
31 infile.open ( " capital " );
32 cout << " \n Contents of capital file : \n ";
33 while(infile)
34 {
35     infile.getline(line, N);
36     cout << line;
37 }
38 infile.close();
39 return 0;

```

When the above code is executed:



```

C:\Users\Pratikk\Documents\file.exe
Contents of Country File -
United States of America United Kingdom South Korea
Contents of capital file :
Washington London Seoul

```

So far we have learnt opening a file using constructor and the function. The opened file can be closed using the function `close()`; which is illustrated in the above examples.

(iii) Reading and Writing Objects to Disks

As we have discussed earlier, one of the disadvantage of I/O system of C is it cannot handle the user – defined data types such as **class objects**. Since the objects are the central objects of C++ programming, naturally the language support the features for writing to and reading from the disk files using objects directly. For this operations, the binary input and output functions **read()** and **write()** are designed to perform the job. The entire structure of an object is handled as a single unit, using the computer internal representation of the data. The point to remember here is that only the data members are copied to the disk files not the member functions.

For example, the function **write()** copies a class object from memory byte by byte with no conversion.

The following program illustrates the working of the **write()** and **read()** functions in the disk files. The length of an object is calculated using the **sizeof()** operator. The length calculated represents the total sum of all the data members of the object.

Program:

```
#include < iostream .h >
#include < fstream.h >
Class ITEM
{
    Char name[20];
    Float price;
    Int code;
Public:
    Void read_data(void);
    Void write_data(void);
};
Void ITEM :: read_data(void)
{
    Cout << "Enter the name of the item: ";
    Cin >> name;
    Cout << "Enter the price of the item:";
    Cin >> price;
    Cout << "Enter the code of the item: ";
    Cin >> code;
}

Void ITEM :: write_data(void)
{
    Cout << setiosflags(ios :: left)
    << setw(10) << name
    << setiosflags(ios :: right)
    << setw(10) << code
    << setprecision(2)
    << setw(10) << cost
    <<endl;
}

Int main()
{
    ITEM item;
    fstream file;
    file.open ( " STOCK.DAT", ios :: in |ios::out );
    cout << "Enter the details for three items \n";
    for( int i=0; i<3; i++)
    {
        item[i].read_data();
        file.write((char *) & item[i], sizeof( item[i]));
    }
    file.seekg(0);

    cout << "\n OUTPUT \n";
    for( i=0; i<3; i++)
    {
        file.read((char *) & item[i], sizeof(item[i]));
        item[i].write_data();
    }
}
```

```
    }  
    file.close();  
    return 0;  
}
```

When the above program is executed:

Enter the details for three items

Enter the name of the item: Hard Disk

Enter the price of the item: 150

Enter the code of the item: 101

Enter the name of the item: CD – ROM

Enter the price of the item: 100

Enter the code of the item: 102

Enter the name of the item: Keyboard

Enter the price of the item: 1000

Enter the code of the item: 106

Output:

HARD DISK	101	150
CD – ROM	102	100
KEYBOARD	106	100

(iv) Binary vs. Character Files

Every file is just a collection of d=series of bytes one after the other, *i.e.*, it contains the numbers between 0 and 255. From the user point of, a file is just a series of bytes.

Generally, every file is referred as a **binary file**. But when a file contains data along with letters, numbers and other symbols which we use while writing a program and if the file contains lines, then it is considered as a **text file**.

Binary file: Most of the software that people use in their daily life consumes and produces binary files. Examples of such type of software are MS Office, Adobe etc., Also a typical computer which is used daily works mostly with the binary files and with very few text files.

Text file: Text files are more readable by humans or at least moderately normal. This file contains more of punctuation symbols like **HTML**, **RTF**, etc., but visible like structure.

Binary file: Matching software is must for the binary files to read and write. *For example*, MP3 file is produced by a sound recorder or an audio editor which cannot play an image viewer or the database software, but can be played in a music player or audio editor.

Text file: The stored is usually line oriented. Each and every line is a separate command, or it may be a list of values that put each item in a different line.

Note: Similarly there are other characteristics of the above said files, which are left as an assignment for the students.

(v) I/O with Multiple Objects

As we already learnt, standard I/O operations are carried out using streams. The same I/O operations are carried on files too. In order to perform I/O operations, the following are three classes which are declared for the same:

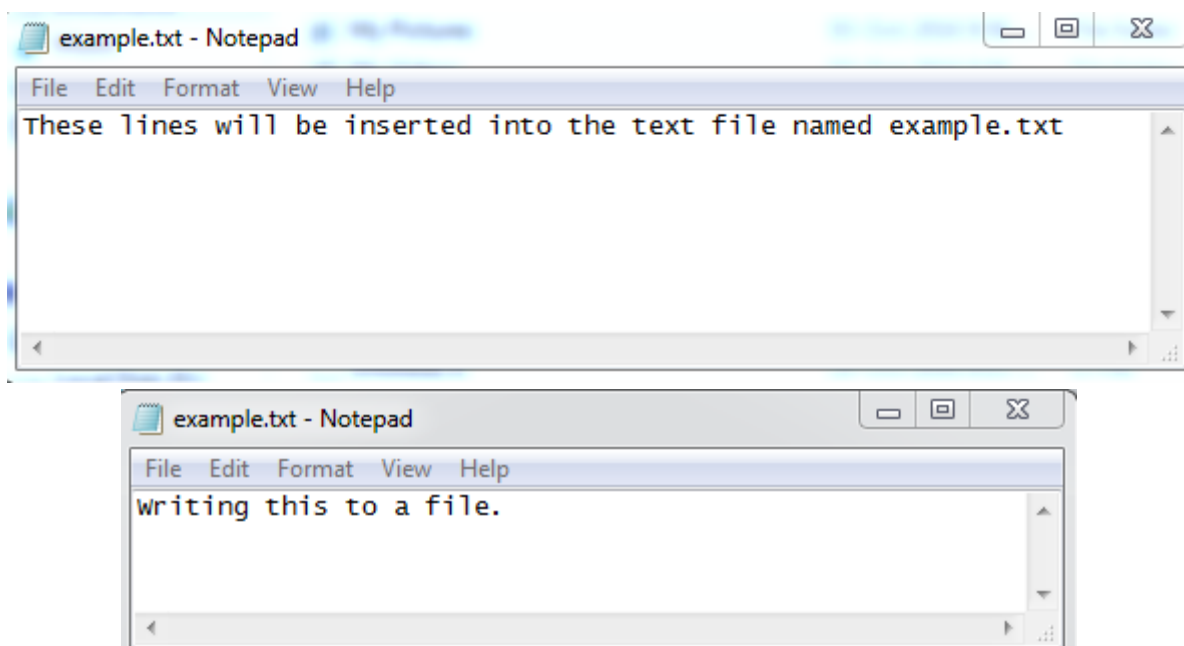
- **ofstream:** stream used for the output to files.
- **ifstream:** stream used for the input from files.
- **fstream:** stream for both input and output operations.

The above classes are derived directly or indirectly from the classes' **istream** and **ostream**. As we know that we have already used objects which are of types of the classes – **cin** is an object of **istream** and **cout** is an object of class **ostream**. Therefore, these classes are used that are related to our files streams.

For example,

```
1  #include<iostream>
2  #include<fstream>
3  using namespace std;
4  int main ()
5  {
6      ofstream myfile;
7      myfile.open ("example.txt");
8      myfile << "Writing this to a file.\n";
9      myfile.close();
10     return 0;
11 }
12 |
```

This code makes a record called example.txt and embeds a sentence into it, similar to how we do with **cout**, yet utilizing the document stream **myfile**.



Following are the step by step operations carried out sequentially:

Opening a file:

The initial operation performed on an object is to link it to a genuine file. This process is named as opening a file. The opened file is shown within a program by a stream and any input or output operations performed on this stream object will also be applied on the physical file which is connected to it.

A file is opened with a stream object using its member function `open` as follows:

```
open(file_name, mode);
```

Where, **file_name** is a string which tells the name of the file to be opened.

mode is an optional parameter with the combination of following flags.

ios::in	Used for input operations.
ios::out	Used for output operations.
ios::binary	Set it in binary mode.
ios::ate	Marks the initial position towards the end of the file If this flag is not marked, the initial position is at the start of the file.
ios::app	All output operations are carried out towards the end of the file, adding the content to the present content of the file.
ios::trunc	If the file is opened for output operations and it already exists, its past content is erased and supplemented by the new one.

All these flags can be combined using the bitwise operator OR (|).

For instance, in the event that we need to open the document `example.bin` in binary mode to include information, we could do it by the accompanying call to member function `open`:

```
ofstream myfile;  
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```



Self-assessment Questions

- 4) Which operator is used to insert the data into file?
- | | |
|-------|-------|
| a) >> | b) << |
| c) < | d) ? |
- 5) Which stream class is used to both read and write on files?
- | | |
|-------------|-------------|
| a) ofstream | b) ifstream |
| c) fstream | d) iostream |
- 6) Which among following is used to open a file in binary mode?
- | | |
|-------------|----------------|
| a) ios::app | b) ios::binary |
| c) ios::in | d) ios::out |

5.2.4 File Pointers

Each and every file object is associated with two integer values which are get pointer and put pointer. These are also called as the **currentgetposition** and the **currentputposition**, or if it is clear it's simply the **currentposition**. These values specify the byte number in the file where writing or reading takes place. When you want to start reading an existing file at the beginning and continue until the end. When writing, you may want to start at the beginning, deleting any existing contents, or at the end, in which case you can open the file with the **ios::app** mode specifies. In some situations the control of the file pointers should be transferred to any location of the file to read from and write to an arbitrary location of the file. The **seekg()** and **tellg()** functions allow you to set and examine the get pointer and the **seekp()** and **tellp()** functions perform these same actions on the put pointer.

(i) Specifying the Position

The **seekg()** function set it to the beginning of the file so that reading would start there. This form of **seekg()** takes one argument, which represents the absolute position in the file. The following diagram illustrates **seekg()** .

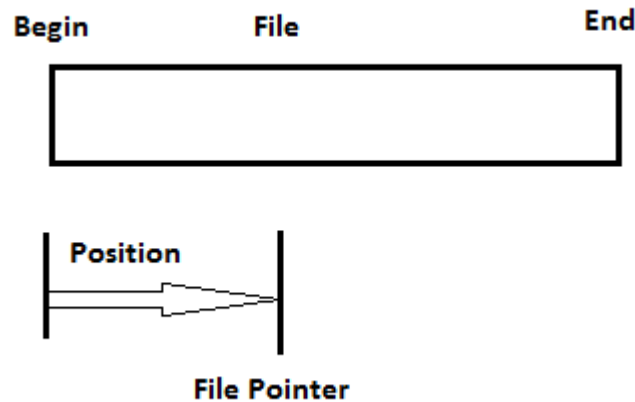


Figure 5.2.7: The `seekg()` Function with One Argument

(ii) Specifying the Offset

The `seekg()` function can be utilized in two ways. We've seen the first, where the single argument represents the position from the beginning of the document. You can likewise utilize it with two arguments, where the first argument represents from a specific position in the file and the second one determines the location from which the previous one that was measured. There are three outcomes for the second argument: `beg` is the start of the file, `cur` is the present pointer position and `end` is the end of the document. The statement `seekp (- 10, ios::end);` for instance, will set the put pointer to 10 bytes before the end of the file is reached.

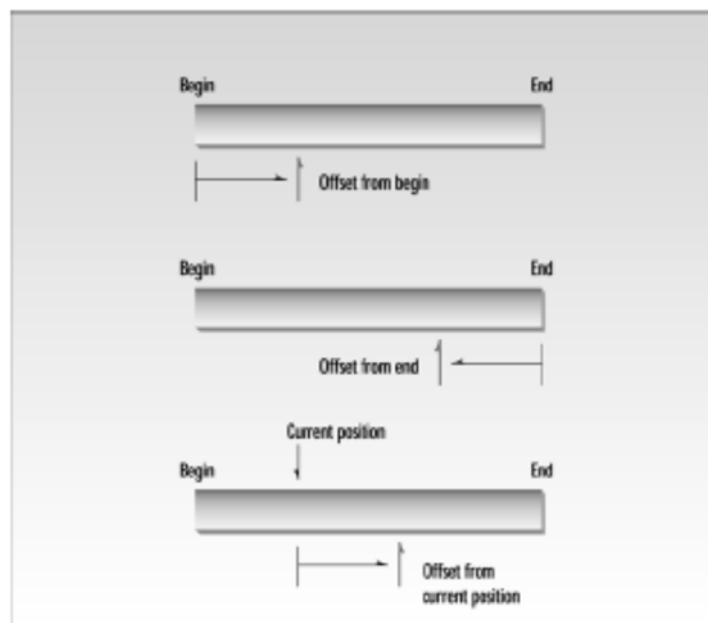


Figure 5.2.8: The `seekg()` Function with Two Arguments

(iii) Seekg() Function

In order to search any random position in a file, the C++ programming language, seekg is a function in the istream library which permits us to do the same thing. This function is characterized for istream class - for ostream class there is one more function Seekp used to avoid clashes in some classes which derives both classes namely istream and ostream, *for example*, istream).

```
istream& seekg ( streampos position );  
istream& seekg ( streamoff offset, ios_base::seekdir dir );
```

- **position** is the new position in the stream buffer. This is an object of type streampos
- **offset** is an integer value of type streamoff representing the offset in the stream's buffer. It is relative to the dir parameter
- **dir** is the seeking direction

It is an object of type ios_base::seekdir that can take any of the following constant values:

1. **ios_base::beg** (offset from the beginning of the stream's buffer)
2. **ios_base::cur** (offset from the current position in the stream's buffer)
3. **ios_base::end** (offset from the end of the stream's buffer)

Specifying positions and offsets are discussed earlier in the above sections. The below program illustrates the working of **seekg()** function.

For example,

```
1  #include <fstream>                                //for file streams  
2  #include <iostream>  
3  using namespace std;  
4  class person                                       //class of persons  
5  {  
6  protected:  
7  char name[80];                                     //person's name  
8  int age;                                           //person's age  
9  public:  
10 void getData()                                     //get person's data  
11 {  
12 cout << "\n Enter name: ";  
13 cin >> name;  
14 cout << "Enter age: ";  
15 cin >> age;  
16 }
```

```

17 void showData(void)                                //display person's data
18 {
19     cout << "\n Name: " << name;
20     cout << "\n Age: " << age;
21 }
22 };
23 int main()
24 {
25     person pers;                                    //create person object
26     ifstream infile;                                //create input file
27     infile.open("GROUP.DAT", ios::in | ios::binary); //open file
28     infile.seekg(0, ios::end);                       //go to 0 bytes from end
29     int endposition = infile.tellg();                //find where we are
30     int n = endposition / sizeof(person);            //number of persons
31     cout << "\nThere are " << n << " persons in file";
32     cout << "\nEnter person number: ";
33     cin >> n;
34     int position = (n-1) * sizeof(person);           //number times size
35     infile.seekg(position);                          //bytes from start
36     //read one person
37     infile.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
38     pers.showData();                                 //display the person
39     cout << endl;
40     return 0;
41 }

```

Here's the output from the program, assuming that the GROUP.DAT file is the same as that just accessed in the example:

```

Terminal
sh-4.3$ g++ -std=c++11 -o main *.cpp
sh-4.3$ main

There are 204522252 persons in file
Enter person number: 3

Name: H@
Age: 1813443840
sh-4.3$ main

There are 204522252 persons in file
Enter person number: 8

Name: H@
Age: -886321536
sh-4.3$

```

(iv) Tellg() Function

The first thing the program does is that it checks out how many persons are in the file. This is done by positioning the **get** pointer at the end of the file with the statement **infile.seekg(0,ios::end);** The **tellg()** function returns the current position of the get pointer. This function is used to return the pointer position to the end of the file, telling the length of

the file in bytes. Finally the program calculates the number of person objects in the file and displays the result. As in the output the user specifies the second object in the file and the program calculates number of bytes in the file using **seekg()**. Finally uses the **read()** to get the specified person data starting from the point. Finally the output is displayed using **showdata()**.



Self-assessment Questions

- 7) A file pointer always contains the address of the file.
 - a) True
 - b) False

- 8) The statement – “f1.write((char*)&obj1, sizeof(obj1));”
 - a) Writes the member functions of obj1 to f1
 - b) Writes the data in obj1 to f1
 - c) Writes the member functions and the data of obj1 to f1
 - d) Writes the address of obj1 to f1

- 9) The tellg() function returns the _____ position of the get pointer.

5.2.5 Command Line Arguments

As we have already used MS-DOS and familiar with the command line arguments, used when invoking a program. There we specify the name of a data file to an application. Let us see the following example,

```
C>wordproc afile.doc
```

Here we invoke a word processor application and the document it will work on at the same time. Here **afile** is a command line argument.

Now let us see how we can read command line arguments in C++. Following example reads and display command – line arguments.

```
// comline.cpp
// demonstrates command-line arguments
```

For example,

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[] )
{
    cout << "\nargc = " << argc << endl;           //number of arguments
    for(int j=0; j<argc; j++)                        //display arguments
        cout << "Argument " << j << " = " << argv[j] << endl;
    return 0;
}
```

And here's a sample interaction with the program:

```
C:\C++BOOK\Chap12>comline uno dos tres
argc = 4
Argument 0 = C:\CPP\CHAP12\COMLINE.EXE
Argument 1 = uno
Argument 2 = dos
Argument 3 = tres
```

In order to read command-line arguments, the `main()` function must itself be given two arguments. The first, `argc` – for argument count, represents the total number of command-line arguments.

In the list of arguments specified, the first command-line argument is always the pathname of the current program. The remaining command-line arguments are those typed by the user; they are delimited by the space character. In the above example they are uno, dos and tres.

The command line arguments are stored in the memory as strings and create an array of pointers to this array. In the above example, the argv is for argument values. Each and every individual strings are accessed through the appropriate pointer.

The first string is argv[0], the second – uno in the above example, is argv[1] and so on. COMLINE accesses the arguments in turn and prints them out in a for loop that uses argc, the number of command-line arguments, as its upper limit.



Did you Know?

When a file is opened for writing only, a new file is created if there is no file of that name. If a file by that name exists already, then its contents are deleted and the file is presented as a clean file.



Self-assessment Questions

- 10) Command-line arguments are accessed through _____ to main()
- 11) Command line arguments application main() function will takes ____ arguments that is,
- | | |
|------|------|
| a) 2 | b) 3 |
| c) 4 | d) 0 |
- 12) argc holds total number of arguments which is passed into main function.
- | | |
|---------|----------|
| a) True | b) False |
|---------|----------|



Summary

- In C++, **fstream** library is used to read and write to/from a file. It consists of three data types they are:
 - **ofstream** represents the output file stream and it is used to create files and to write information to files,
 - **ifstream** represents the input file stream and is used to read information from files.
 - **fstream** represents the file stream and it has the capabilities of both **ofstream** and **ifstream**.
- The console operator namely **cin** and **cout** are used to accept the inputs and display the output through the console respectively.
- Manipulators are a group of functions which are utilized to control the output formats of the console operators.
- Files in C++ are connected with objects of different classes, commonly **ofstream** for output, **ifstream** for input and **fstream** for both input and output. Member functions of these or base classes are utilized to perform I/O operations. Such operators and functions as **<<**, **put()** and **write()** are utilized for output, while **>>**, **get()** and **read()** are utilized for input.
- The **read()** and **write()** functions work in binary mode, so whole objects can be put on to the disk regardless of what kind of information they contain. Single objects can be put saved, as an array or other data.
- The extraction operator **>>** and the insertion operator **<<** are overloaded so that they work with programmer-defined data types.



Terminal Questions

1. Explain console I/O Operator.
2. Brief about formatted and unformatted I/O operators.
3. Explain file handling in C++ with example.
4. Illustrate reading and writing an object from and to the disk by opening and closing a file.
5. Demonstrate programs on `tellg()` function and `seekg()` function.
6. Elaborate on command line arguments.



Answer Keys

Self-assessment Questions	
Question No.	Answer
1	Ios
2	d
3	Ifstream, ofstream, fstream
4	b
5	c
6	d
7	b
8	b
9	arguments
10	a
11	a
12	a



Activity

Activity Type: Online

Duration: 30 Minutes

Description:

1. Develop and execute a program that uses functions to perform the following operations:
 - a. To copy contents of one file into another file
 - b. To replace a word with other word in a given file
 - c. To count the no of occurrences of a word in a given file

Bibliography



e-References

- cplusplus.com, (2016). Input/output with files. Retrieved 7 April 2016, from <http://www.cplusplus.com/doc/tutorial/files/>
- eskimo.com, (2016). File Pointers and fopen. Retrieved 7 April 2016, from <https://www.eskimo.com/~scs/cclass/notes/sx12a.html>
- perlmaven.com, (2016). Beginner perlmaven tutorial. Retrieved 7 April 2016, from <http://perlmaven.com/perl-tutorial>
- nayuki.com, (2016). What are binary and text files?. Retrieved 7 April 2016, from <https://www.nayuki.io/page/what-are-binary-and-text-files>



External Resources

- Balaguruswamy, E. (2008). *Object Oriented Programming with C++*. Tata McGraw Hill Publications.
- Lippman. (2006). *C++ Primer (3rd ed.)*. Pearson Education.
- Robert, L. (2006). *Object Oriented Programming in C++ (3rd ed.)*. Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). *C++ completer (1st ed.)*. Tata McGraw Hill.
- Strousstrup. (2005). *The C++ Programming Language (3rd ed.)*. Pearson Publications



Video Links

Topic	Link
Manipulators	https://www.youtube.com/watch?v=ezzmiX_pJAQ
Opening and closing a file	https://www.youtube.com/watch?v=jVifX_wlrO4
Command line arguments	https://www.youtube.com/watch?v=LU5nQUxaoiU



Notes:

