Unit

# 9

# Working with Files

## 1. Introduction

All the programs we have seen so far use input only from the keyboard, and output only to the screen. In many programs, we need to input large amount of data. In such cases, the data has to be entered every time the program is executed. Moreover, both, the input as well as the output are lost once the computer is turned off. If we want the input data or the program output in a permanent form, we have to store it on the secondary memory device like magnetic tapes, CDs or hard disk so that the data can be used whenever required. The data can be stored in *files* on these devices. The program can read input data from a file and also send its output to a file.

In this chapter, we will see how data can be stored and read from files and what are the operations performed on files.

*There are two types of files*

i. **Text Files:** These files store data in the form of characters. Even numeric data is stored as individual digits. There are additional special characters to signify end of file (EOF) and end of line.

ii. **Binary Files:** These files store data in the form of their internal representation in computer memory. In binary files, a character occupies 1 byte, integer 2 bytes and float 4 bytes. Thus they occupy same amount of disk space as memory space. There are no special characters to indicate end of file or end of line.

# 2. File Streams

In C++, all I/O operations are performed using streams. A stream is a sequence of bytes that flows from an I/O device to the program. A file stream is an interface between the program and the file. The input stream reads data from file and the output stream writes data to the file. The following diagram shows a schematic representation of file input/output using streams.

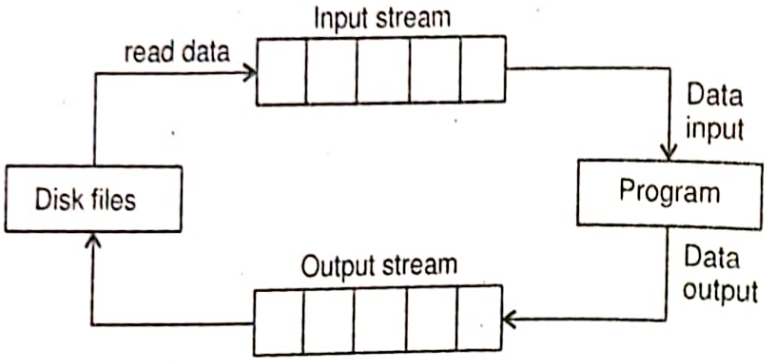


**Figure 9.1: File input/ output stream**

C++ has a rich set of built-in stream classes for console based I/O as well as file based I/O. The following diagram shows the class hierarchy of the stream classes.

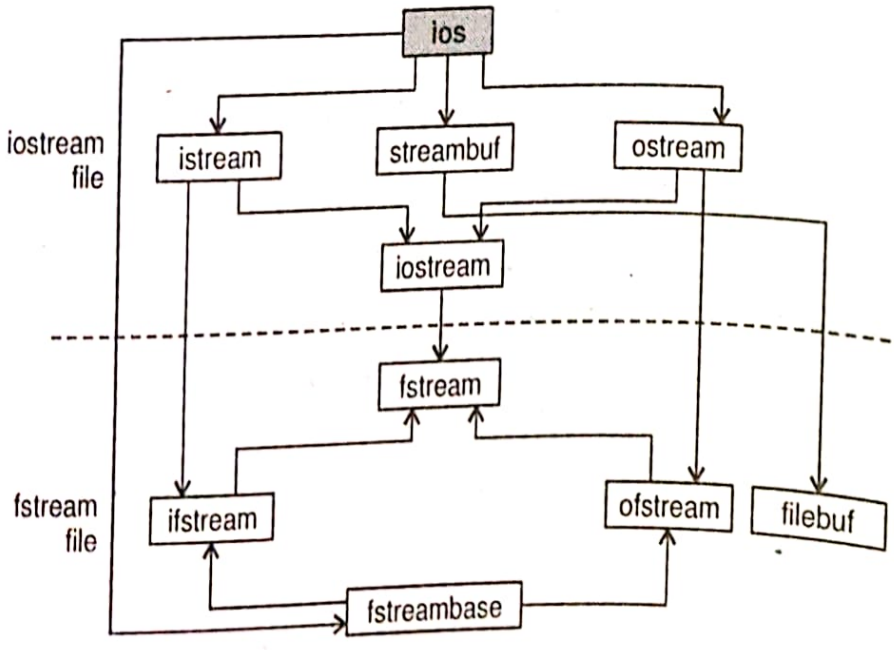


**Figure 9.2: C++ stream class hierarchy**

### File stream Classes

There are five file stream classes in "fstream.h" which are used for file operations. These are explained in the table below:

**Table 9.1: File stream classes in C++**

| Class | Purpose |
|---|---|
| fstreambase | Provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream classes. Contains open() and close() functions. |
| filebuf | Its purpose is to set the file buffers to read and write. Also contains close() and open() as members. |
| ifstream | Provides input operations. Contains open() with default input mode. Inherits get(), getline(), read(), seekg() and tellg() functions from istream. |
| ofstream | Provides output operations. Contains open() with default output mode. Inherits put(), seekg(), tellp() and write() functions from ostream. |
| fstream | Provides support for simultaneous input and output operations. Inherits all the functions from istream and ostream through iostream. Considers open() with default input mode. |

### File stream methods

The file stream classes have several methods which are used to perform operations on file objects. These methods are discussed in the following sections.

## 3. File Operations

To perform input-output operations on files, we have to create objects of the file stream classes described above. Three stream classes are commonly used for this purpose:

1. To create an input stream, create an object of the class **ifstream**.
2. To create an output stream, create an object of the class **ofstream**.
3. Stream for performing both input and output operations must be object of class **fstream**.

*For example,*

```
1.  ifstream in;          // input
2.  ofstream out;         // output
3.  fstream io;           // input and output
```

Once the file stream object is created, we can perform file operations. The operations that are performed on a file are: opening a file, closing a file, reading from a file, writing to a file, detecting end of a file and performing random access to a file.

## 3.1 Opening a File

In order to perform any operation on a file, the file has to be first opened. A file can be opened in two ways:

1. By using the constructor of the stream class.
2. By using member function open() of the class.

1. **Opening Files Using Constructor:** Each file stream class has a constructor which takes the filename as parameter. Hence, to open a file:

   i. Create a file stream object.

   ii. Initialize the file object with the filename.

   *For example,*

   1. To open a file "student.txt" for reading:
      ```
      ifstream in("student.txt"); //input only
      ```
   2. To open a file "student.txt" for writing:
      ```
      ofstream out("student.txt"); //output only
      ```
   3. To open a file "modify.txt" for reading and writing:
      ```
      fstream inout("modify.txt"); //input and output
      ```

### File Opening Modes

In the above examples, we have passed only one argument to the constructor i.e. the filename. However, we can also pass two arguments. The first is the name of the file to be opened. The second argument specifies the file mode.

The general form of the constructor is:

```
filestream-class stream-object("filename");
                    or
filestream-class stream-object("filename", mode);
```

The second argument *mode* (called file mode parameter) specifies the purpose for which the file is opened. These modes are defined as static constants in the ios class.

Table 9.2: File opening modes in C++

| Modes for File Open | |
|---|---|
| **Mode Name** | **Operation** |
| ios::app | Appends data to the end of file. |
| ios::ate | When first opened, positions file at end-of-file (ate stand for at-end ). |
| ios::in | Opens file for reading. |
| ios::nocreate | Open fails if the file does not exist. (No longer supported) |
| ios::noreplace | If file exists, open for output fails. |
| ios::out | Opens file for writing. |
| ios::trunc | Truncates file if it already exists. |
| ios::binary | Opens file in binary mode. |

We can combine two or more modes or flags by using bitwise operator OR (|). *For example,* to open a file for output and position it at the end of existing data, we can write the statement as follows:

```
ofstream out("outfile.txt",ios::out|ios::ate);
```

If we do not provide the value for the mode, then it will use the default values shown in the following table.

Table 9.3: Default file opening modes

| Class | Default mode |
|---|---|
| ofstream | ios::out \| ios::trunc |
| ifstream | ios::in |
| fstream | ios::in \| ios::out |

2. **Opening a File Using open():** Another method to open a file is to use the function open().There are two steps to be followed in this case:

   i. Create a file stream object.

   ii. Call the open function by passing filename as parameter.

*The syntax* to call the open function is given below:

```
stream-object.open("filename");
                or
stream-object.open("filename",mode);
```

*Example: To open a file example.txt for writing*

```
ofstream out;                    //create stream for output
out.open("example.txt");         //connect stream to file
```

***Which method to use?***

If we want to use the stream object with just one file, it is better to use the constructor. But if we want to use the same stream object on several files, then we should use the open function.

*For example,*

```
1.  ifstream in("a.txt");        //constructor
2.  ifstream in;                 //using open function
    in.open("file1.txt");
    ...
    in.open("file2.txt");
```

## ▶Checking for Open Failure

We should always check whether the file was successfully opened before we perform any operations on the file. It may happen that the file could not be opened for some reason. *For example*, we may try to open a read-only file in the output mode. C++ provides several mechanisms for checking for success or failure of a file-open operation. The following table lists these mechanisms to check whether a file object named "file" was successfully opened or not.

**Table 9.4: Checking file open success or failure**

| Test | Success | Failure |
|---|---|---|
| if ( file ) | true | false |
| if (! file) | false | true |
| if( file.good() ) | true | false |
| if( !file.good() ) | false | true |
| if( file.fail() ) | false | true |
| if( !file.fail() ) | true | false |

*Example:*

```
ifstream in;
in.open("a.txt");
if(in.fail())
{  cout << "Sorry, the file couldn't be opened!\n";
     exit(1);
}
     ...
}
```

or

```
if(!in)
{
   ....
}
```

## 3.2 Closing a File

The member function close() is used to close a file after use. It is called automatically by the destructor. However it can be called explicitly. Closing a file releases any resources that may have been allotted for processing the file. The close() member function does not take any arguments nor does it return any value.

The *syntax* for close is given below:

```
stream-object.close();
```

*Example*

```
ifstream in("a.txt");
//perform operations
in.close();
```

## 3.3 Detecting End-of-File

The eof() member function is used to check whether the file pointer has reached at the end of a file or not. If it is successful, eof() function returns a nonzero, otherwise returns a zero.

*Syntax*

```
stream-object.eof()
```

*Example*

```
while(!in.eof())
{
   //read contents
}
```

In addition to **eof()**, we can also use the stream-object directly to check whether the eof has been reached or not. *For example*:

```
while(!in)
{
   //perform operations
}
```

## 3.4 Reading / Writing a Character

There are two methods to read and write characters from a file.

1. Using the insertion and extraction operators on the stream object.
2. Using member functions get() and put().

1. **Using Insertion and Extraction Operators:** The << and >> operators can be used to read from and write data to files.

*For example,*

i. Write a character 'a' to a file stream named "out":

```
out<<'a';
```

ii. Write an integer 20, character 'B' and string "Hello" to a file object "out":

```
out<<20<<'B'<<"Hello";
```

The following program accepts characters from the user till the user enters EOF (Ctrl+Z) and writes them to a file named a.txt. The program then reads the characters from the file and displays them on the screen.

**Program : Read and write characters using << and >> operators**

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main()
{
  ofstream out("a.txt");
  if(!out)
  {
     cout<<"File opening error";
     exit(1);
  }
  char ch;
  cout<<"Enter the characters - Ctrl Z to end"<<endl;
  while((cin>>ch))
   out<<ch;
 out.close();
 ifstream in("a.txt"); //open the file for reading
 if(!in)
 {
    cout<<"File opening error";
    exit(1);
 }
```

```
cout<<"The contents of a.txt are:\n";
while(!in.eof())
{
  in>>ch;
  cout<<ch;
}
in.close();
return 0;
}
```

**Output**

Enter the characters - Ctrl Z to end

abcdefghijk012345 Ctrl+Z

The contents of a.txt are:
abcdefghijk012345

Note Using cin, we cannot read spaces, tabs or new line characters from the console.

Let us write another program to see the use of << and >> operators on files. The file 'cities.txt' contains names of cities and their STD codes. We will read the contents of this file and search for a specific city name and display the STD code.

**Program : Search file contents**

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main()
{
  char city[20], std[10], str[20];
  int flag=0;
  ifstream in("cities.txt");
  if(!in)
  {
     cout<<"File opening error";
     exit(1);
  }
  cout<<"Enter the name of the city to be searched :"<<endl;
  cin>>str;
  while(!in.eof())
  {
     in>>city>>std;
      if(strcmp(city, str)==0)
      {  cout<<"City :"<<city<<endl<<"STD code : "<<std<<endl;
          flag=1;break;
     }
  }
  in.close();
  if(flag==0)
     cout<<"Sorry, city name not found";
  return 0;
   }
```

**Output**

Enter the name of the city to be searched:
MUMBAI
City: MUMBAI
STD code: 022

2. **Using get() and put():** The stream classes have two member functions get() and put() fo input and output respectively.

i. get(): This member function reads a character from a file stream.

*Syntax*

```
stream-object.get()
```

*Example:*

```
ifstream in("a.txt");
char ch;
while(!in.eof())
{
   ch = in.get();
   cout<<ch;
}
```

ii. put() : This member function is used to write a character to an output stream.

*Syntax:*

```
stream-object.put(character)
```

*Example:*

```
ofstream out("a.txt");
char str[] = "C++ Program";
for(int i=0; str[i]!= '\0'; i++)
out.put(str[i]);  // write a character to stream
```

**Program : Use of get and put functions**

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main()
{
  ofstream out("a.txt");
  char string[80] = "Program using get() and put()", ch;
  if(!out)
  {  cout<<"File opening error";
     exit(1);
  }
 for(int i=0; string[i]!='\0';i++)
out.put(string[i]);
out.close();
ifstream in("a.txt");
cout<<"The contents of a.txt are:\n";
while(!in.eof())
{ ch = in.get();
   cout << ch; }
in.close();
return 0;
}
```

***Output***

The contents of a.txt are:
Program using get() and put()

## 3.5 Reading and Writing Block of Data

In the previous section we saw how to read and write a single character using files. In many cases, we may want to read and write a set of data values to a file. C++ provides two functions: **read()** and **write()** to do so. These functions deal with data in the binary form i.e. data is stored in the file in the same manner that it is stored in the internal memory. The type of file is a binary file.

*Syntax*

```
stream-object.read((char *) address, size);
stream-object.write((char *) address, size);
```

Where *address* is the address of the memory block where the data is stored, *size* is an integer that specifies the number of characters to be read from the file or written to the file.

***Examples:***

1. To write a float variable to file "numbers"

```
float f = 45.678;
ofstream out("numbers.dat", ios::binary);
out.write((char *)&f, sizeof(f));
```

2. To write an array of 5 integers to a file named "numbers"

```
int arr[5] = {10,20,30,40,50};
ofstream out("numbers.dat", ios::binary);
out.write((char *)arr, sizeof(arr));
```

3. To read 3 integers from a file named "numbers"

```
int a[3];
ifstream in("numbers.dat", ios::binary);
in.read((char *)a, 3*sizeof(int));
```

**Note** If the source or destination of data is an array, then we need not use "&" to obtain the address.

The following program writes an array A of 10 integers to a file named "numbers" and reads 5 integers into an array called B.

**Program : Using read() and write()**

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main()
{
  fstream file;
```

```
int A[10] = {10,20,30,40,50,60,70,80,90,100};
file.open("numbers", ios::in|ios::out|ios::binary);
if(!file)
{ cout<<"File opening error";
   exit(1);
}
file.write((char *)A, sizeof(A));
   //write array A to file
file.seekg(0); //move to the beginning
int B[5];
file.read((char *)B, 5*sizeof(int));
   //read 5  integers
cout<<"The contents of array B are:\n";
for(int i=0;i<5;i++)
   cout<<B[i]<<" ";
file.close();
return 0;
}
```

***Output***

The contents of array B are:
10 20 30 40 50

Only a single call to read() or write() is necessary to read or write the entire array. Each individual element need not be read or written separately.

## 3.6 Reading and Writing Objects

C++ programs basically deal with classes and objects. Hence, in many cases, we require objects to be stored in files so that we can read them later and perform operations. We can use read() and write() functions to read and write class objects to binary files. When we write an object to a file, only the data members will be written to the file. Member functions are not stored in the file.

To read and write an object using a file, the following syntax is to be used:

```
stream-object.read((char *) & object-name, size);
stream-object.write((char *) & object-name, size);
```

***Examples:***

1. To write an object named s1 of class student to file "data"

```
student s1(10, "abc");
ofstream out("data", ios::binary);
out.write((char *)&s1, sizeof(s1));
```

2. To read a single object of type student into object named s2 from file "data"

```
student s2;
ifstream in("data", ios::binary);
in.read((char *)&s2, sizeof(s2));
```

3. To write an array of 3 objects of the type student into file "data"

```
student s[3];
...
ofstream out("data", ios::binary);
out.write((char *)s, sizeof(s));
```

Let us write a program which writes details of three students into a file named "data" and then reads the contents from the file into another array and displays it on the screen.

**Program : Read and write objects**

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
class student
{
  int rno;
  char name[20];
  public:
    student(int r=0, char *str="") //constructor
    {
      rno = r;
      strcpy(name, str);
    }
    void display()
    {
      cout<<" Roll number = "<<rno<<endl;
      cout<<" Name = "<<name<<endl;
    }
};
int main()
{ student s1(1,"A");
  student s2(2,"B");
  student s3(3,"C");
  fstream file;
  file.open("data", ios::in|ios::out|ios::binary);
  if(!file)
```

```
{
      cout<<"File opening error";
      exit(1);
}
file.write((char *)&s1, sizeof(s1));
file.write((char *)&s2, sizeof(s2));
file.write((char *)&s3, sizeof(s3));
file.seekg(0);
student arr[3];   //an array of 3 students
file.read((char *)arr, sizeof(arr));
cout<<"The student details are:\n";
for(int i=0;i<3;i++)
arr[i].display();
file.close();
return 0;
}
```

***Output***

```
The student details are:
Roll Number = 1
Name = A
Roll Number = 2
Name = B
Roll Number = 3
Name = C
```

We can modify the above program to search for a specific student in the file on the basis of the roll number. The file contains n student objects.

**Program : Search object in file**

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
class student
{
  int rno;
  char name[20];
  public:
    void accept()
    {
      cout<<"Enter roll number :"; cin>>rno;
      cout<<"Enter name :"; cin>>name;
    }
    void display()
    {
      cout<<" Roll number = "<<rno;
      cout<<" Name = "<<name<<endl<<endl;
    }
    int getroll()  //accessor function
    {
      return rno;
    }
```

```
};
int main()
{
  student s;
  int n,r;
  ofstream outfile;
  outfile.open("data", ios::binary);
  if(!outfile)
  {
     cout<<"File opening error";
     exit(1);
  }
  cout<<"How many objects :";
  cin>>n;
  for(int i=0;i<n;i++)
  {
     s.accept();
     outfile.write((char *)&s, sizeof(s));
  }
  outfile.close();
  ifstream infile("data");
  cout<<"Enter the roll number to be searched :";
  cin>>r;
  int flag = 0;
  while(!infile.eof())
  {
     infile.read((char *)&s, sizeof(s));
     if(s.getroll() == r)
     {
        cout<<"Student found :";
        s.display();flag = 1;
        break;
     }
  }
  if(flag==0)
    cout<<"Student not found ";
  infile.close();
  return 0;
}
```

**Output**

```
How many objects: 3
Enter roll number: 15
Enter name: ABC
Enter roll number: 20
Enter name: DEF
Enter roll number: 10
Enter name: XYZ
Enter the roll number to be
searched: 20
Student found:
Roll Number = 20Name = DEF
```

# 4. File Updation with Random Access

Each file has two pointers associated with it. One is the input pointer or *get* pointer and the other is the output pointer or *put* pointer. The input pointer is used to read the contents of a given file location and the output pointer is used to write to a given file location. As operations are performed on the file, these pointers are moved.

There are some default actions that are associated with these stream pointers:

1. When we open a file for reading, the get pointer is set to the beginning of the file.
2. When we open a file for writing, the put pointer is positioned at the beginning.
3. When a file is opened in the append or the at-end mode, put pointer is positioned at the end of the file.

C++ provides functions to move the pointers and also to find out the current position of the pointer. These functions are listed below.

1. **tellg() and tellp():** These two member functions give the current position of the get and put pointer respectively. They take no parameters and return an integer representing the current position of the pointer. The current position is the offset (in number of bytes) with reference to the beginning of the file.

   *Example 1*

```
ifstream in("a.txt");
int n = in.tellg();
cout<<n;
```

   The above *example* will give 0 since the get pointer is at the beginning of the file.

   *Example 2*

```
ofstream out("a.txt", ios::app);
int n = in.tellp();
cout<<n;
```

   In the above *example*, the put pointer will be positioned at the end of the file and hence, n will give the number of bytes in the file i.e. the file size.

2. **seekg() and seekp():** These functions allow us to move the *get* and *put* stream pointers to any desired location in the file. The seekp() function moves the put pointer while the seekg() function moves the get pointer. Both functions are overloaded with two different prototypes:

```
seekg(pos_type position);
seekp(pos_type position);
```

   Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. *position* refers to the number of bytes that the pointer is moved.

The second prototype takes two arguments: the first is the number of bytes by which the pointer is to be moved. The second is the reference with respect to which the pointer is to be moved.

```
seekg(offset, ref_position);
seekp(offset, ref_position);
```

The reference position can be one of the following

**Table 9.5: File reference positions**

| Ref_position | Meaning |
|---|---|
| ios::beg | offset specified from the beginning of the file |
| ios::cur | offset specified from the current position of the pointer |
| ios::end | offset specified from the end of the file |

*Example 1: To read the 21st character from the file*

```
ifstream in("a.txt");
in.seekg(20) ;    // move the get pointer to the 21st byte
cout<<in.get();
```

*Example 2: To read the 5th student object from file "stud.dat"*

```
ifstream in("stud.dat", ios::binary);
in.seekg(4*sizeof(student)) ;
student s;
in.read((char *)&s , sizeof(student));
```

*Example 3: To insert a character '*' at the 5th position in file "a.txt"*

```
ofstream out("a.txt", ios::ate);
out.seekp(4*sizeof(char)) ;
out.put('*');
```

The following *examples* illustrate how the get pointer can be moved in various ways using a stream object named "file".

**Table 9.6: Examples of moving the "get" pointer**

| Statement | Action |
|---|---|
| `file.seekg(m)` | Move by m bytes from the beginning of the file |
| `file.seekg(0);` | Move to the beginning of the file |
| `file.seekg(0, ios::end)` | Move to the end of the file |
| `file.seekg(m, ios::beg)` | Move by m bytes from the beginning of the file |
| `file.seekg(m, ios::cur)` | Move by m bytes from the current position |
| `file.seekg(-m, ios::cur)` | Move backward by m bytes from the current position |
| `file.seekg(-m, ios::end)` | Move backward by m bytes from the end of file |

e following *example* uses the member functions just seen to obtain the size of a text file and also plays the file in the reverse order:

**Program : Obtaining file size**

```
#include<iostream.h>
#include<fstream.h>
int main()
{
  long n;
  char str[] ="ABCDEFGHIJKL";
  fstream file("a.txt",ios::in|ios::out);
  file<<str;
  file.seekg(0,ios::end);
  n = file.tellg();
  cout<<"The file size = "<<n <<" bytes"<<endl;
  cout<<"File contents in reverse order are : ";
  while(n)
  {
     file.seekg(n-1,ios::beg);
     cout<<(char)file.get();
     n--;
  }
  file.close();
  return 0;
}
```

**Output**

The file size = 12 bytes
File contents in reverse order are :
LKJIHGFEDCBA

## 5. Overloading << and >> Operators

In chapter 6, we have seen how the insertion and extraction operators can be overloaded. We can overload these operators to read and write class objects from files.

*Example:*

```
ofstream out("student.dat",ios::binary);
student s(20, "abc");
out<<s;   //write object to file
```

The syntax of overloading the operators is as follows:

*Declaration Syntax:*

```
friend  ofstream& operator<<(ofstream&, const class-name&);
friend  ifstream& operator>>(ifstream&, class-name&);
```

*Definition Syntax:*

```
ofstream& operator<<(ofstream& out, const class-name& obj)
{
  //write data members of obj to stream object out
  return out;
}
ifstream& operator>>(ifstream& in, class-name& obj)
{
  //read data members of obj from stream object in
  return in;
}
```

Let us now overload these operators for the student class.

**Program : Overloading << and >> operators**

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>
class student
{
  int rno;
  char name[20];
  public:
    student(){ }
    student(int r, char *nm)
    {
      rno=r; strcpy(name, nm);
    }

  friend ofstream& operator<<(ofstream&, const student&);
  friend ifstream& operator>>(ifstream&, student&);
  void display();
};
  ofstream& operator<<(ofstream& out, const student& obj)
  {
    out<<obj.rno<<"\t"<<obj.name<<"\n";
    //Write roll no and name
    return out;
  }
  ifstream& operator>>(ifstream& in, student& obj)
  {
    in>>obj.rno>>obj.name; //Read roll no and name
    return in;
  }
  void student::display()
  {
    cout<<"Roll No ="<<rno<< " Name = "<<name<<endl;
  }
```

```
void main()
{
   student s1(1, "abc"), s2(2, "def");
   ofstream out("students.dat", ios::binary);
   out<<s1<<s2;
   out.close();
   ifstream in("students.dat",ios::binary);
   in>>s1>>s2;
   s1.display();
   s2.display();
   in.close();
}
```

**Output**

Roll No = 1 Name = abc
Roll No = 2 Name = def

## Solved Examples

**1. Program to read contents of a text file and count number of characters, words and lines in the file. The program also performs a word search operation on the file.**

```
#include<fstream.h>
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include<string.h>
class TextFile
{
  char filename[20];
  fstream file;
  int chars, words, lines;
  public:
  TextFile(char *str);
   void countwords();
   void search(char *word);
};
TextFile::TextFile(char *str)
{
  strcpy(filename, str);
  chars=0; words=0; lines=0;
}
void TextFile::countwords()
{
  file.open(filename,ios::in);
  if(!file)
  {
```

```
    cout<<"File opening error ";
    return;
  }
  int INWORD=0;  char ch;
  while(!file.eof())
  {
      ch=file.get();
      chars++;
      if(ch=='\n') lines++;
      switch(ch)
      {
        case ' ':
        case '\t':
        case '\n':
             if(INWORD==1)
               words++;
             INWORD=0;
             break;
       default: INWORD=1;
     }
  }
  file.close();
  cout<<"Characters = "<<chars<<endl;
  cout<<"Lines = "<<lines<<endl;
  cout<<"Words = "<<words<<endl;
}
void TextFile::search(char *word)
{
  char str[20];
  file.open(filename,ios::in);
  while(!file.eof())
  {
     file>>str;
     if(strcmp(str,word)==0)
     {       cout<<"word found";
             return;
     }
   }
   cout<<"Word not found in file ";
   file.close();
}
void main()
{
```

```
  char word[20];
  TextFile t("a.txt");
  t.countwords();
  cout<<"Enter word to be searched :";
  cin>>word;
  t.search(word);
}
```

**2. A program to change the case of every alphabet of a text file.**

```
// converting a lower case to upper case letter
#include<stream.h>
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include<ctype.h>
void main()
{
   char fname1[10], fname2[10];
   ofstream outfile;
   ifstream infile;
   char ch, uch;
   cout <<"enter a file name to be copied ? \n";
   cin >> fname;
   cout <<"new file name ?\n";
   cin >> fname2;
   infile.open(fname1);
   if(infile.fail())
   { cerr <<"unable to create a file \n";
        exit(1);
   }
   outfile.open(fname2);
   if(outfile.fail())
   {
      cerr <<"unable to create a file \n";
      exit(1);
   }
   while(!infile.eof())
   { ch = (char)infile.get();
      if(isalpha(ch))
```

```
        if(isupper(ch))
            ch = tolower(ch);
        else
            ch = toupper(ch);
        outfile.put(ch);
    }
    infile.close();
    outfile.close();
}
```

# EXERCISES

## A. Fill in the Blanks

1. _____ class can be used for both read/write C++ file I/O operations.
2. Opening a file in ios::out mode also opens it in the _____ mode by default.
3. The read() and write() functions handle data in _____form.
4. The _____ function is used to check whether a file pointer is reached at the end of a file or not.
5. _____function is used to read an alphanumeric character from a specified file.

## B. State True or False

1. The header file ifstream is derived from the base class istream.
2. ofstream is used to write a stream of objects in a file.
3. The fail( ) function is used to check whether a file pointer is reached the end of a file character or not.
4. The input pointer is also called as put pointer.
5. For opening a file we must first create a file stream.
6. We can add data to an existing file by opening in write mode.

## C. Subjective Questions

1. Explain the various file opening modes in C++.
2. Explain how random access to files can be achieved.

3. Explain the ways to check for successful file open.
4. Explain how insertion and extraction operators can be overloaded for files.
5. Write a note on tellg() ands tellp().
6. Explain the use of seekg and seekp with examples.

## D. Programming Exercises

1. Write a program in C++ to read a file and to
   i. Display the contents of the file on to the screen,
   ii. Display the number of characters and
   iii. The number of lines in the file.
2. Write a program in C++ to read a file and to display the contents of the file.
3. Write a program in C++ to merge two files into a one file heading.
4. Write a program in C++ to read students record such as name, sex, roll number, height and weight from the specified file and to display in a sorted order (name is the key for sorting).
5. There are 100 records present in a file with each record containing a 6-character item code, a 20-character item name and an integer quantity. Write a program to read these records, arrange them in the ascending order and write them in the same file overwriting the earlier records.

## Answers

**A.**

| | | | | | |
|---|---|---|---|---|---|
| 1. | fstream | 2. | ios::trunc | 3. | binary |
| 4. | eof() | 5. | get() | | |

**B.**

| | | | | | |
|---|---|---|---|---|---|
| 1. | True | 2. | True | 3. | False |
| 4. | False | 5. | False | 6. | False |