Manipulating Strings

15

Key Concepts

C-strings | The string class | Creating string objects | Manipulating strings | Relational operations on strings | Comparing strings | String characteristics | Swapping strings

15.1 INTRODUCTION

A string is a sequence of characters. We know that C++ does not support a built-in string type. We have used earlier null-terminated character arrays to store and manipulate strings. These strings are called *C-strings* or *C-style strings.* Operations on C-strings often become complex and inefficient. We can also define our own string classes with appropriate member functions to manipulate strings. This was illustrated in Program 7.4 (Mathematical Operation of Strings).

ANSI standard C++ now provides a new class called **string.** This class improves on the conventional C-strings in several ways. In many situations, the string objects may be used like any other built-in type data. Further, although it is not considered as a part of the STL, **string** is treated as another container class by C++ and therefore all the algorithms that are applicable for containers can be used with the **string** objects. For using the **string** class, we must include **<string>** in our program.

The **string** class is very large and includes many constructors, member functions and operators. We may use the constructors, member functions and operators to achieve the following:

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string objects
- Comparing string objects
- Adding string objects
- Accessing characters in a string
- Obtaining the size of strings
- Many other operations

Table 15.1 gives prototypes of three most commonly used constructors and Table 15.2 gives a list of important member functions. Table 15.3 lists a number of operators that can be used on **string** objects.

Table 15.1 *Commonly used string constructors*

Constructor	Usage
String();	For creating an empty string
String(const chat *str);	For creating a string object from a null-terminated string
String(const string & str);	For creating a string object from other string object

Table 15.2 *Important functions supported by the string class*

Function	Task
append()	Appends a part of string to another string
Assign()	Assigns a partial string
at()	Obtains the character stored at a specified location
Begin()	Returns a reference to the start of a string
capacity()	Gives the total elements that can be stored.
compare()	Compares string against the invoking string
empty()	Returns true if the string is empty; Otherwise returns false
end()	Returns a reference to the end of a string
erase()	Removes characters as specified
find()	Searches for the occurrence of a specified substring
insert()	Inserts characters at a specified location
length()	Gives the number of elements in a string
max size()	Gives the maximum possible size of a string object in a give system
replace()	Replace specified characters with a given string
resize()	Changes the size of the string as specified
size()	Gives the number of characters in the string
swap()	Swaps the given string with the invoking string

Table 15.3 *Operators for string objects*

Operator	Meaning
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal

(Contd.)

Table 15.3 (Contd.)

>	Greater than
>=	Greater than or equal
[]	Subscription
<<	Output
>>	Input

15.2 CREATING (string) OBJECTS

We can create **string** objects in a number of ways as illustrated below:

```
string s1;              // Using constructor with no argument
string s2("xyz");       // Using one-argument constructor
s1 = s2;                // Assigning string objects
s3 = "abc" + s2         // Concatenating strings
cin >> s1;              // Reading through keyboard (one word)
getline(cin, s1);       // Reading through keyboard a line of text
```

The overloaded + operator concatenates two string objects. We can also use the operator += to append a string to the end of a string. Examples:

```
s3 += s1;               // s3 = s3 + s1
s3 += "abc";            // s3 = s3 + "abc"
```

The operators << and >> are overloaded to handle input and output of string objects. Examples:

```
cin >> s2;              // Input to string object (one word)
cout << s2;             // Displays the contents of s2
getline(cin, s2);       // Reads embedded blanks
```

Note *Using **cin** and >> operator we can read only one word of a string while the **getline()** function permits us to read a line of text containing embedded blanks.*

Program 15.1 demonstrates the several ways of creating string objects in a program.

Program 15.1 Creating String Objects

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
        // Creating string objects
        string s1;              // Empty string object
        string s2(" New");      // Using string constant
```

(Contd.)

```
string s3(" Delhi");

// Assigning value to string objects
s1 = s2;                    // Using string object
cout << "S1 = " << s1 << "\n";

// Using a string constant
s1 = "Standard C++";
cout << "Now S1 = " << s1 << "\n";

// Using another object
string s4(s1);
cout << "S4 = " << s4 << "\n\n";

// Reading through keyboard
cout << "ENTER A STRING \n";
cin >> s4;                  // Delimited by blank space
cout << "Now S4 = " << s4 << "\n\n";

// Concatenating strings
s1 = s2 + s3;
cout << "S1 finally contains: " << s1 << "\n";

return 0;
}
```

The output of Program 15.1 would be:

```
S1 = New
Now S1 = Standard C++
S4 = Standard C++
ENTER A STRING
COMPUTER CENTRE
Now S4 = COMPUTER

S1 finally contains: New Delhi
```

15.3 MANIPULATING STRING OBJECTS

We can modify contents of **string** objects in several ways, using the member functions such as **insert()**, **replace()**, **erase()**, and **append()**. Program 15.2 demonstrates the use of some of these functions.

Program 15.2 Modifying String Objects

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
```

(Contd.)

```
    string s1("12345");
    string s2("abcde");

    cout << "Original Strings are: \n";
    cout << "S1: " << s1 << "\n";
    cout << "S2: " << s2 << "\n\n";

    // Inserting a string into another
    cout << "Place S2 inside S1 \n";
    s1.insert(4,s2);
    cout << "Modified S1: " << s1 << "\n\n";

    // Removing characters in a string
    cout << "Remove 5 Characters from S1 \n";
    s1.erase(4,5);
    cout << "Now S1: " << s1 << "\n\n";

    // Replacing characters in a string
    cout << "Replace Middle 3 Characters in S2 with S1 \n";
    s2.replace(1,3,s1);
    cout << "Now S2: " << s2 << "\n";

    return 0;
}
```

The output of Program 15.2 given below illustrates how strings are manipulated using string functions.

```
Original Strings are:
S1: 12345
S2: abcde

Place S2 inside S1
Modified S1: 1234abcde5

Remove 5 Characters from S1
Now S1: 12345

Replace Middle 3 Characters in S2 with S1
Now S2: a12345e
```

Note *Analyze how arguments of each function used in this program are implemented.*

15.4 RELATIONAL OPERATIONS

A number of operators that can be used on strings are defined for **string** objects (Table 15.3). We have used in the earlier examples the operators = and + for creating objects. We can also apply the relational operators listed in Table 15.3. These operators are overloaded and can be used to compare **string** objects. The **compare()** function can also be used for this purpose.

Program 15.3 shows how these operators are used.

Program 15.3 Relational Operations on String Objects

```
#include<iostream>
#include<string.h>

using namespace std;

int main()
{
string s1("ABC");
string s2("XYZ");
string s3 = s1 + s2;

if(s1!=s2)
    cout<<s1<<" is not equal to "<<s2<<"\n";

if(s1>s2)
    cout<<s1<<" is greater than "<<s2<<"\n";
else
    cout<<s2<<" is greater than "<<s1<<"\n";

if(s3==s1+s2)
    cout<<s3<<" is equal to "<<s1+s2<<"\n\n";

int x = s1.compare(s2);
if(x==0)
    cout<<s1<<" = "<<s2<<"\n";

else if(x>0)
    cout<<s1<<" > "<<s2<<"\n";
else
    cout<<s1<<" < "<<s2<<"\n";

return 0;
}
```

The output of Program 15.3 would be:

```
ABC is not equal to XYZ
XYZ is greater than ABC
ABCXYZ is equal to ABCXYZ

ABC < XYZ
```

15.5 STRING CHARACTERISTICS

Class **string** supports many functions that could be used to obtain the characteristics of strings such as size, length, capacity, etc. The size or length denotes the number of elements currently stored in a given string. The capacity indicates the total elements that can be stored in the given string. Another characteristic is the *maximum size* which is the largest possible size of a string object that the given system can support. Program 15.4 illustrates how these characteristics are obtained and used in an application.

Program 15.4 Obtaining String Characteristics

```
#include <iostream>
#include <string>

using namespace std;

void display(string &str)
{
        cout << "Size = " << str.size() << "\n";
        cout << "Length = " << str.length() << "\n";
        cout << "Capacity = " << str.capacity() << "\n";
        cout << "Maximum Size = " << str.max_size() << "\n";
        cout << "Empty: " << (str.empty() ? "yes" : "no");
        cout << "\n\n";
}

int main()
{
        string str1;

        cout << "Initial status: \n";
        display(str1);

        cout << "Enter a string (one word) \n";
        cin >> str1;
        cout << "Status now: \n";
        display(str1);

        str1.resize(15);
        cout << "Status after resizing: \n";
        display(str1);
        cout << "\n";

        return 0;
}
```

The output of Program 15.4 would be:

```
Initial status:
Size = 0
Length = 0
Capacity = 0
Maximum Size = 4294967293
Empty: yes
Enter a string (one word)
INDIA
Status now:
Size = 5
Length = 5
Capacity = 31
```

```
Maximum Size = 4294967293
Empty: no

Status after resizing:
Size = 15
Length = 15
Capacity = 31
Maximum Size = 4294967293
Empty: no
```

The size and length of 0 indicate that the string **str1** contains no characters. The size and length are always the same. The **str1** has a capacity of zero initially but its capacity has increased to 31 when a string is assigned to it. The maximum size of a string in this system is 4294967293. The function **empty()** returns **true** if **str1** is empty; otherwise **false.**

15.6 ACCESSING CHARACTERS IN STRINGS

We can access substrings and individual characters of a string in several ways. The **string** class supports the following functions for this purpose:

```
at()              for accessing individual characters
substr()          for retrieving a substring
find()            for finding a specified substring
find_first_of()   for finding the location of first occurrence of the
                  specified character(s)
find_last_of()    for finding the location of last occurrence of the
                  specified character(s)
```

We can also use the overloaded [ ] operator (which makes a **string** object look like an array) to access individual elements in a string. Program 15.5 demonstrates the use of some of these functions.

Program 15.5 Accessing and Manipulating Characters

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
        string s("ONE TWO THREE FOUR");
        cout << "The string contains: \n";
        for(int i=0;i<s.length();i++)
            cout << s.at(i);                    // Display one character
        cout << "\nString is shown again: \n";
        for(int j=0;j<s.length();j++)
            cout << s[j];

        int x1 = s.find("TWO");
        cout << "\n\nTWO is found at: " << x1 << "\n";
```

(Contd.)

```
    int x2 = s.find_first_of('T');
    cout << "\nT is found first at: " << x2 << "\n";
    int x3 = s.find_last_of('R');
    cout << "\nR is last found at: " << x3 << "\n";

    cout << "\nRetrieve and print substring TWO \n";

    cout << s.substr(x1,3);
    cout << "\n";

    return 0;
}
```

The output of Program 15.5 would be:

```
The string contains:
ONE TWO THREE FOUR
String is shown again:
ONE TWO THREE FOUR

TWO is found at: 4

T is found first at: 4

R is last fount at: 17

Retrieve and print substring TWO
TWO
```

We can access individual characters in a string using either the member function **at()** or the subscript operator [ ]. This is illustrated by the following statements:

```
cout << s.at(i);
cout << s[i];
```

The statement

```
int x1 = s.find("TWO");
```

locates the position of the first character of the substring "TWO". The statement

```
cout << s.substr(x1,3);
```

finds the substring "TWO". The first argument **x1** specifies the location of the first character of the required substring and the second argument gives the length of the substring.

15.7 COMPARING AND SWAPPING

The **string** supports functions for comparing and swapping strings. The **compare()** function can be used to compare either two strings or portions of two strings. The **swap()** function can be used for swapping the contents of two **string** objects. The capabilities of these functions are demonstrated in Program 15.6.

Program 15.6 Comparing and Swapping Strings

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        string s1("Road");
        string s2("Read");
        string s3("Red");
        cout << "s1 = " << s1 << "\n";
        cout << "s2 = " << s2 << "\n";
        cout << "s3 = " << s3 << "\n";

        int x = s1.compare(s2);
        if(x == 0)
            cout << "s1 == s2" << "\n";
        else if(x > 0)
            cout << "s1 > s2" << "\n";
        else
            cout << "s1 < s2" << "\n";

        int a = s1.compare(0,2,s2,0,2);
        int b = s2.compare(0,2,s1,0,2);
        int c = s2.compare(0,2,s3,0,2);
        int d = s2.compare(s2.size()-1,1,s3,s3.size()-1,1);

        cout << "a = " << a << "\n" << "b = " << b << "\n";
        cout << "c = " << c << "\n" << "d = " << d << "\n";

        cout << "\nBefore swap: \n";
        cout << "s1 = " << s1 << "\n";
        cout << "s2 = " << s2 << "\n";
        s1.swap(s2);
        cout << "\nAfter swap: \n";
        cout << "s1 = " << s1 << "\n";
        cout << "s2 = " << s2 << "\n";

        return 0;
}
```

The output of Program 15.6 would be:

```
s1 = Road
s2 = Read
s3 = Red
s1 > s2
a = 1
b = -1
c = 0
d = 0
```

```
Before swap:
s1 = Road
s2 = Read

After swap:
s1 = Read
s2 = Road
```

The statement

```
int x = s1.compare(s2);
```

compares the string **s1** against **s2** and **x** is assigned 0 if the strings are equal, a positive number if **s1** is *lexicographically* greater than **s2** or a negative number otherwise.

The statement

```
int a = s1.compare(0,2,s2,0,2);
```

compares portions of **s1** and **s2**. The first two arguments give the starting subscript and length of the portion of **s1** to compare to **s2**, that is supplied as the third argument. The fourth and fifth arguments specify the starting subscript and length of the portion of **s2** to be compared. The value assigned to **a** is 0, if they are equal, 1 if substring of **s1** is greater than the substring of **s2**, -1 otherwise.

The statement

```
s2.swap(s2);
```

exchanges the contents of the strings **s1** and **s2**.

SUMMARY

- Manipulation and use of C-style strings become complex and inefficient. ANSI C++ provides a new class called **string** to overcome the deficiencies of C-strings.
- The string class supports many constructors, member functions and operators for creating and manipulating string objects. We can perform the following operations on the strings:
  - Reading strings from keyboard
  - Assigning strings to one another
  - Finding substrings
  - Modifying strings
  - Comparing strings and substrings
  - Accessing characters in strings
  - Obtaining size and capacity of strings
  - Swapping strings
  - Sorting strings

Working with Files

11

Key Concepts

Console-user interaction | Input stream | Output stream | File stream classes | Opening a file with open() | Opening a file with constructors | End-of-file detection | File modes | File pointers | Sequential file operations | Random access files | Error handling | Command-line arguments

11.1 INTRODUCTION

Many real-life problems handle large volumes of data and, in such situations, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of *files*. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

1. Data transfer between the console unit and the program.
2. Data transfer between the program and a disk file.

This is illustrated in Fig. 11.1.

We have already discussed the technique of handling data communication between the console unit and the program. In this chapter, we will discuss various methods available for storing and retrieving the data from files.

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as *input stream* and the one that receives data from the program is known as *output stream*. In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file. This is illustrated in Fig. 11.2.

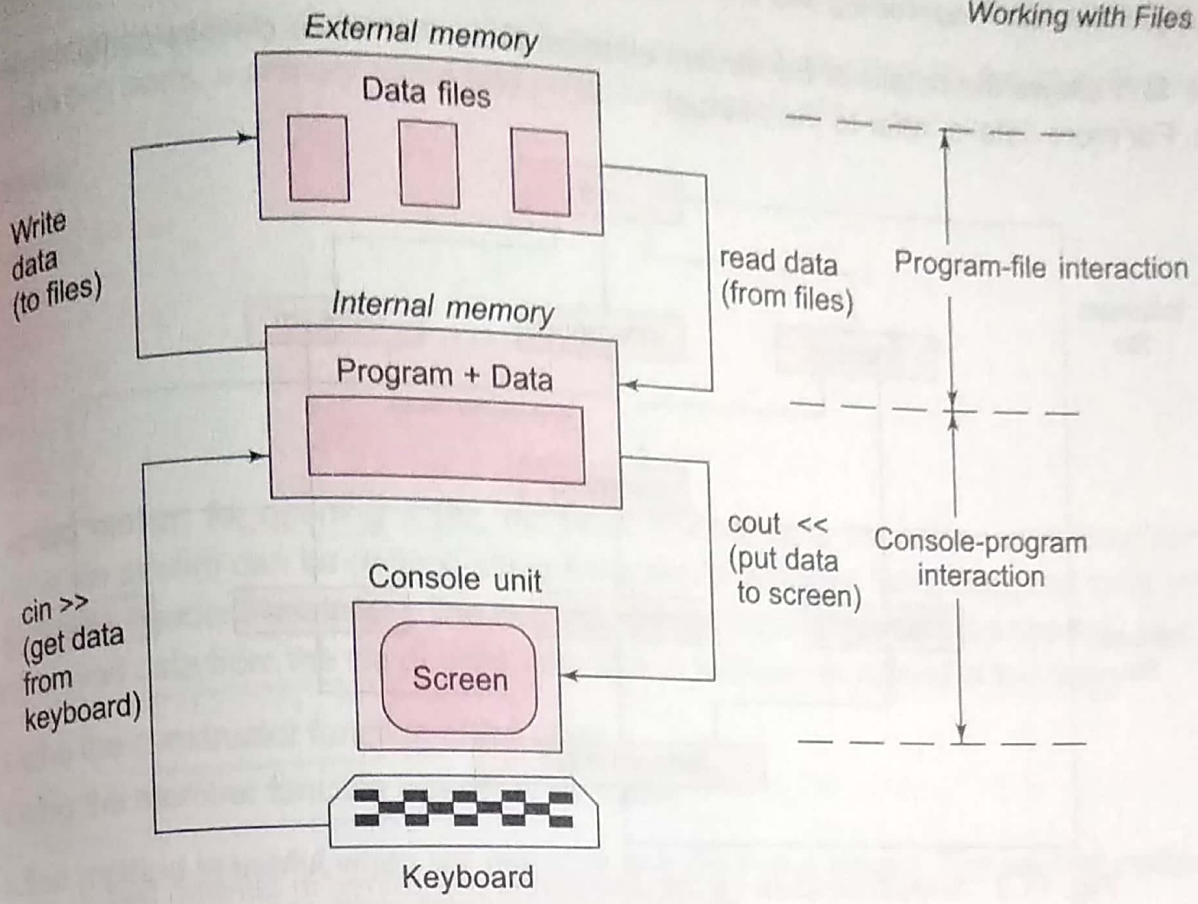

Fig. 11.1 *Consol-program-file interaction*

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

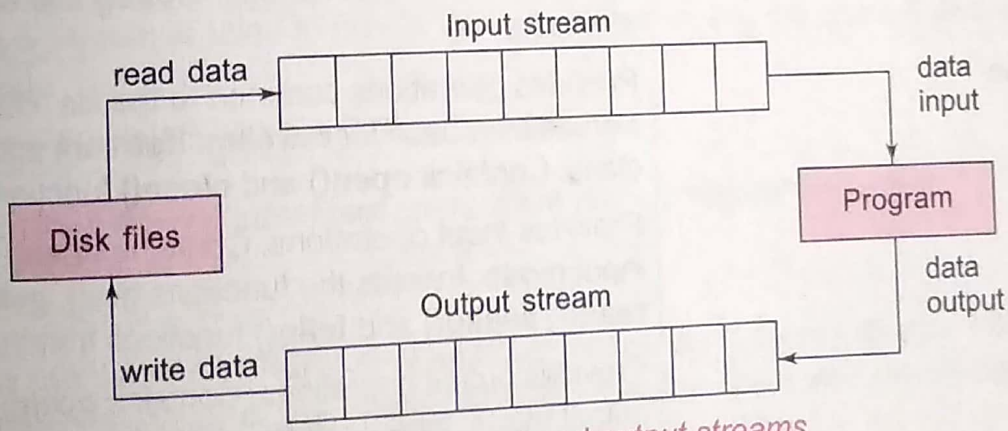

Fig. 11.2 *File input and output streams*

11.2 CLASSES FOR FILE STREAM OPERATIONS

The I/O system of C++ contains a set of classes that define the file handling methods. These include **ifstream, ofstream** and **fstream**. These classes are derived from **fstreambase** and from the corresponding *iostream* class as shown in Fig. 11.3. These classes, designed to manage the disk files, are declared in *fstream* and therefore, we must include this file in any program that uses files.

Table 11.1 shows the details of file stream classes. Note that these classes contain many mo features. For more details, refer to the manual.

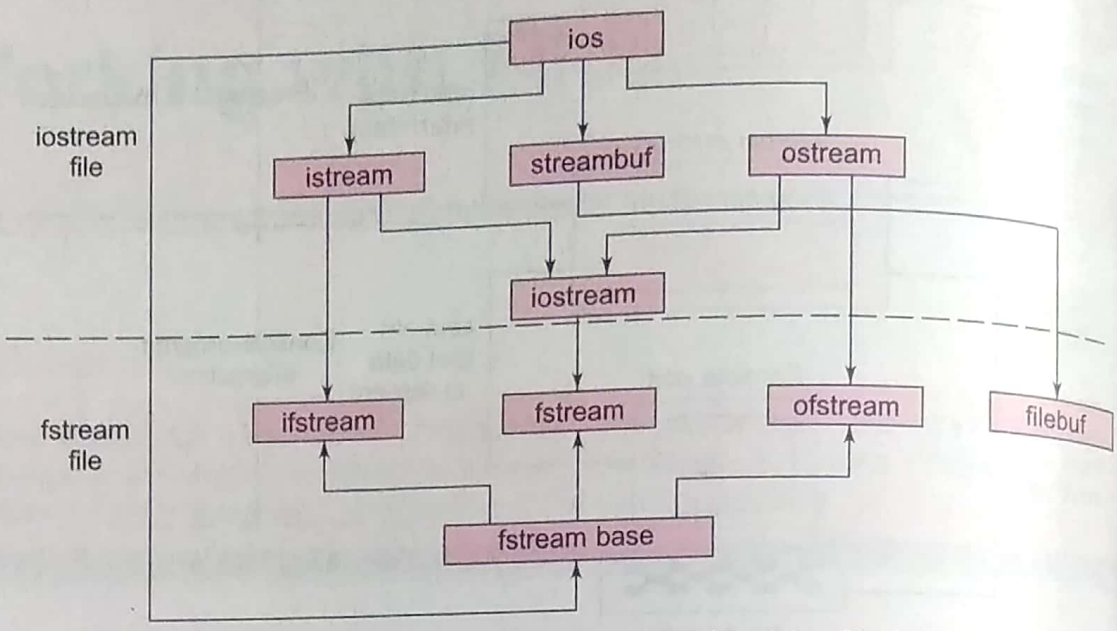

Fig. 11.3 *Stream classes for file operations (contained in fstream file)*

Table 11.1 *Details of file stream classes*

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to the file streams. Serves as a base for **fstream**, **ifstream** and **ofstream** class. Contains **open()** and **close()** functions.
ifstream	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get()**, **getline()**, **read()**, **seekg()** and **tellg()** functions from **istream**.
ofstream	Provides output operations. Contains **open()** with default output mode. Inherits **put()**, **seekp()**, **tellp()**, and **write()**, functions from **ostream**.
fstream	Provides support for simultaneous input and output operations. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

11.3 OPENING AND CLOSING A FILE

If we want to use a disk file, we need to decide the following things about the file and its intended us

1. Suitable name for the file
2. Data type and structure
3. Purpose
4. Opening method

The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension.

Examples:

```
Input.data
Test.doc
INVENT.ORY
student
salary
OUTPUT
```

As stated earlier, for opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes **ifstream**, **ofstream**, and **fstream** that are contained in the header file *fstream*. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

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

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

Opening Files Using Constructor

We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is to say, 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 filename.

For example, the following statement opens a file named "results" for output:

```
ofstream outfile("results");                    // output only
```

This creates **outfile** as an **ofstream** object that manages the output stream. This object can be any valid C++ name such as **o_file**, **myfile** or **fout**. This statement also opens the file results and attaches it to the output stream **outfile**. This is illustrated in Fig. 11.4.

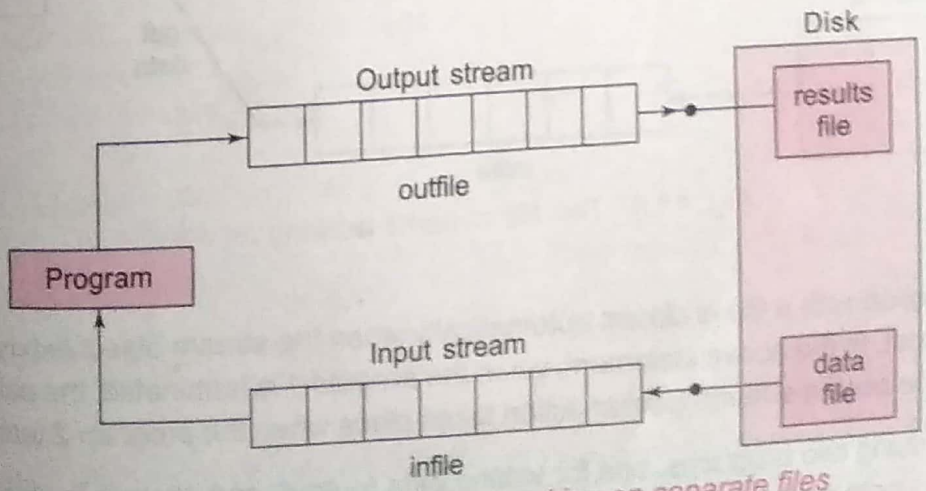

Fig. 11.4 Two file streams working on separate files

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

```
ifstream infile("data");          // input only
```

The program may contain statements like:

```
outfile << "TOTAL";
outfile << sum;
infile >> number;
infile >> string;
```

We can also use the same file for both reading and writing data as shown in Fig. 11.5. The programs would contain the following statements:

```
Program1
    .....
    .....
    ofstream outfile("salary");        // creates outfile and
                                          connects
                                       // "salary" to it
    .....
    .....
Program2
    .....
    .....
    ifstream infile("salary");         // creates infile and connects
                                       // "salary" to it
    .....
    .....
```

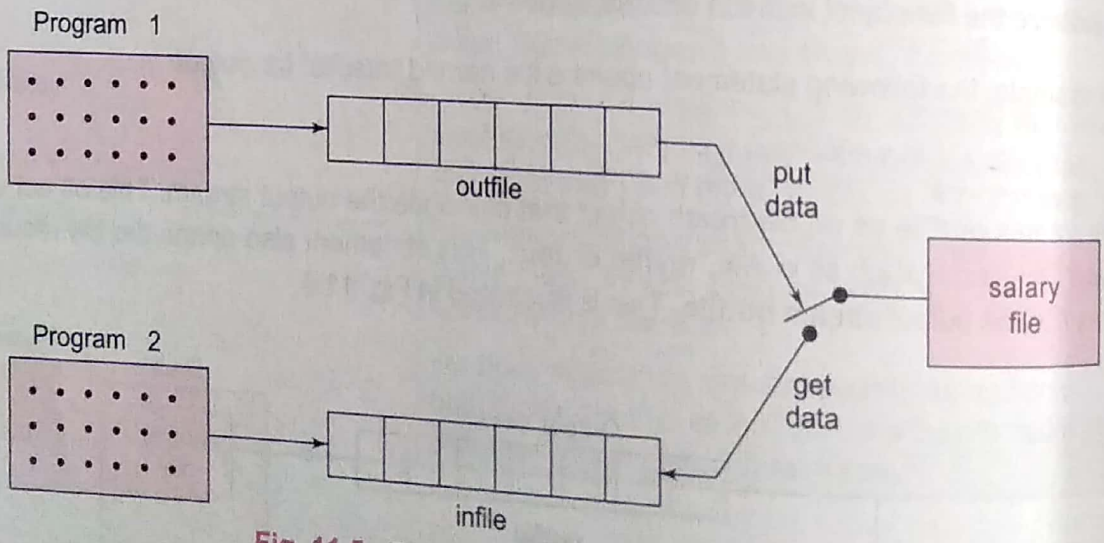

Fig. 11.5 *Two file streams working on one file*

The connection with a file is closed automatically when the stream object expires (when the program terminates). In the above statement, when the *program1* is terminated, the **salary** file is disconnected from the **outfile** stream. Similar action takes place when the *program 2* terminates.

Instead of using two programs, one for writing data (output) and another for reading data (input), we can use a single program to do both the operations

```
.....
.....
        outfile.close();
        ifstream infile("salary");      // Disconnect salary from outfile
                                        // and connect to infile
.....
.....
        infile.close();                 // Disconnect salary from infile
```

Although we have used a single program, we created two file stream objects, **outfile** (to put data into the file) and **infile** (to get data from the file). Note that the use of a statement like

```
outfile.close();
```

disconnects the file salary from the output stream **outfile**. Remember, the object **outfile** still exists and the **salary** file may again be connected to **outfile** later or to any other stream. In this example, we are connecting the **salary** file to **infile** stream to read data.

Program 11.1 uses a single file for both writing and reading the data. First, it takes data from the keyboard and writes it to the file. After the writing is completed, the file is closed. The program again opens the same file, reads the information already written to it and displays the same on the screen.

Program 11.1 Working with Single File

```
// Creating files with constructor function

#include <iostream.h>
#include <fstream.h>

int main()
{
        ofstream outf("ITEM");          // connect ITEM file to outf

        cout << "Enter item name:";
        char name[30];
        cin >> name;                    // get name from key board and

        outf << name << "\n";           // write to file ITEM

        cout << "Enter item cost:";
        float cost;
        cin >> cost;                    // get cost from key board and

        outf << cost << "\n";           // write to file ITEM

        outf.close();                   // Disconnect ITEM file from outf

        ifstream inf("ITEM");           // connect ITEM file to inf

        inf >> name;                    // read name from file ITEM
```

(Contd.)

```
                                        // read cost from file ITEM
    inf >> cost;
    cout << "\n";
    cout << "Item name:" << name << "\n";
    cout << "Item cost:" << cost << "\n";
                                        // Disconnect ITEM from inf
    inf.close();

    return 0;
}
```

The output of Program 11.1 would be:

```
Enter item name:CD-ROM
Enter item cost:250

Item name:CD-ROM
Item cost:250
```

Caution *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. We shall discuss later how to open an existing file for updating it without losing its original contents.*

Opening Files Using open()

As stated earlier, the function **open()** can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

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

Example:

```
ofstream outfile;
outfile.open("DATA1");                  // Create stream (for output)
.....                                   // Connect stream to DATA1
.....
outfile.close();
outfile.open("DATA2");                  // Disconnect stream from DATA1
.....                                   // Connect stream to DATA2
.....
outfile.close();
.....
.....                                   // Disconnect stream from DATA2
```

The previous program segment opens two files in sequence for writing the data. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time. See Program 11.2 and Fig. 11.6.

Program 11.2 Working with Multiple Files

```
// Creating files with open() function

#include <iostream.h>
#include <fstream.h>

int main()
{
    ofstream fout;
    fout.open("country");                    // create output stream
                                             // connect "country" to it
    fout << "United States of America\n";
    fout << "United Kingdom\n";
    fout << "South Korea\n";

    fout.close();                            // disconnect "country" and

    fout.open("capital");                    // connect "capital"

    fout << "Washington\n";
    fout << "London\n";
    fout << "Seoul\n";

    fout.close();                            // disconnect "capital"

    // Reading the files
    const int N = 80;                        // size of line
    char line[N];

    ifstream fin;                            // create input stream
    fin.open("country");                     // connect "country" to it

    cout <<"contents of country file\n";

                                             // check end-of-file
    while(fin)
    {
            fin.getline(line, N);            // read a line
            cout << line ;                   // display it
    }
                                             // disconnect "country" and
    fin.close();                             // connect "capital"
    fin.open("capital");

    cout << "\nContents of capital file \n";

    while(fin)
    {
            fin.getline(line, N);
            cout << line ;
```

(Contd.)

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

The output of Program 11.2 would be:

```
Contents of country file
United States of America
United Kingdom
South Korea

Contents of capital file
Washington
London
Seoul
```

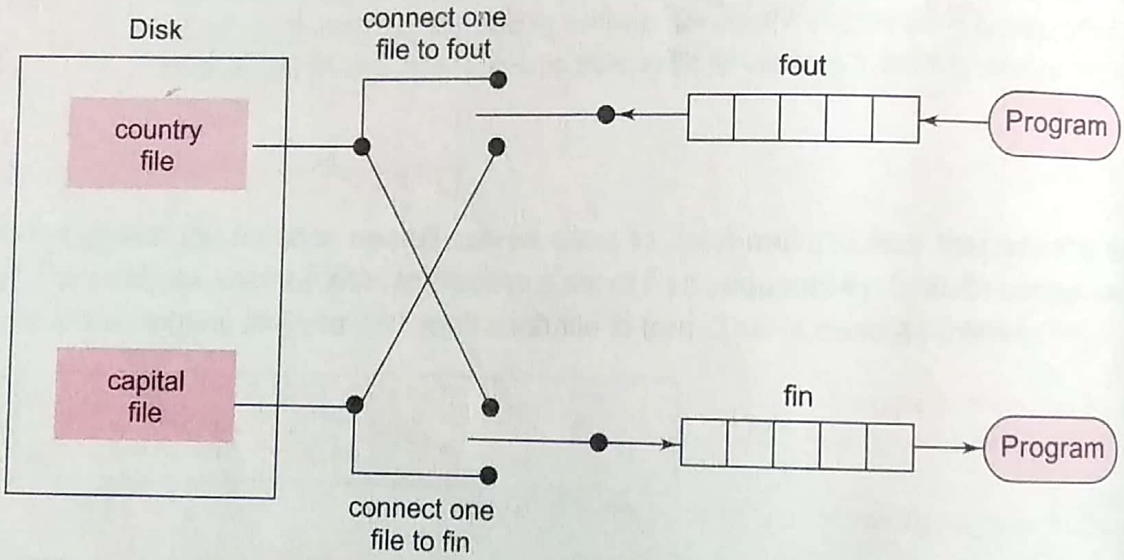

Fig. 11.6 *Streams working on multiple files*

At times we may require to use two or more files simultaneously. For example, we may require to merge two sorted files into a third sorted file. This means, both the sorted files have to be kept open for reading and the third one kept open for writing. In such cases, we need to create two separate input streams for handling the two input files and one output stream for handling the output file. See Program 11.3.

Program 11.3 Reading from Two Files Simultaneously

```
// Reads the files created in Program 11.2

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
                                    // for exit() function
```

```
int main()
{
        const int SIZE = 80;
        char line[SIZE];

        ifstream fin1, fin2;                    // create two input streams
        fin1.open("country");
        fin2.open("capital");

        for(int i=1; i<=10; i++)
        {
           if(fin1.eof() != 0)
           {
             cout << "Exit from country \n";
             exit(1);
           }

           fin1.getline(line, SIZE);
           cout << "Capital of "<< line ;

           if(fin2.eof() != 0)
           {
             cout << "Exit from capital\n";
             exit(1);
           }

           fin2.getline(line,SIZE);
           cout << line << "\n";
        }
    return 0;
}
```

The output of Program 11.3 would be:

```
Capital of United States of America
Washington
Capital of United Kingdom
London
Capital of South Korea
Seoul
```

11.4 DETECTING END-OF-FILE

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file. This was illustrated in Program 11.2 by using the statement

```
while(fin)
```

An **ifstream** object, such as **fin**, returns a value of 0 if any error occurs in the file operation including the end-of-file condition. Thus, the **while** loop terminates when **fin** returns a value of zero on reaching the end-of-file condition. Remember, this loop may terminate due to other failures as well. (We will discuss other error conditions later.)

There is another approach to detect the end-of-file condition. Note that we have used the following statement in Program 11.3:

```
if(fin1.eof() != 0) {exit(1);}
```

eof() is a member function of **ios** class. It returns a non-zero value if the end-of-file(EOF) condition is encountered, and a zero, otherwise. Therefore, the above statement terminates the program on reaching the end of the file.

11.5 MORE ABOUT OPEN(): FILE MODES

We have used **ifstream** and **ofstream** constructors and the function **open()** to create new files as well as to open the existing files. Remember, in both these methods, we used only one argument that was the filename. However, these functions can take two arguments, the second one for specifying the file mode. The general form of the function **open()** with two arguments is:

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

The second *argument mode* (called file mode parameter) specifies the purpose for which the file is opened. How did we then open the files without providing the second argument in the previous examples?

The prototype of these class member functions contain default values for the second argument and therefore they use the default values in the absence of the actual values. The default values are as follows:

```
ios::in for ifstream functions meaning open for reading only.
ios::out for ofstream functions meaning open for writing only.
```

The *file mode* parameter can take one (or more) of such constants defined in the class **ios**. Table 11.2 lists the file mode parameters and their meanings.

Table 11.2 *File mode parameters*

Parameter	Meaning
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

Note

1. Opening a file in **ios::out** mode also opens it in the **ios::trunc** mode by default.
2. Both **ios::app** and **ios::ate** take us to the end of the file when it is opened. The difference between the two parameters is that the **ios::app** allows us to add data to the end of the file only, while **ios::ate** mode permits us to add data or to modify the existing data anywhere in the file. In both the cases, a file is created by the specified name, if it does not exist.
3. The parameter **ios::app** can be used only with the files capable of output.
4. Creating a stream using **ifstream** implies input and creating a stream using **ofstream** implies output. So in these cases, it is not necessary to provide the mode parameters.
5. The **fstream** class does not provide a mode by default and therefore, we must provide the mode explicitly when using an object of **fstream** class.
6. The mode can combine two or more parameters using the bitwise OR operator (symbol |) as shown below:

```
fout.open("data", ios::app | ios:: nocreate)
```

This opens the file in the append mode but fails to open the file if it does not exist.

11.6 FILE POINTERS AND THEIR MANIPULATIONS

Each file has two associated pointers known as the *file pointers*. One of them is called the input pointer (*or get pointer*) and the other is called the output pointer (*or put pointer*). We can use these pointers to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

Default Actions

When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start. Similarly, when we open a file in write-only mode, the existing contents are deleted and the output pointer is set at the beginning. This enables us to write to the file from the start. In case, we want to open an existing file to add more data, the file is opened in 'append' mode. This moves the output pointer to the end of the file (i.e., the end of the existing contents). See Fig. 11.7.

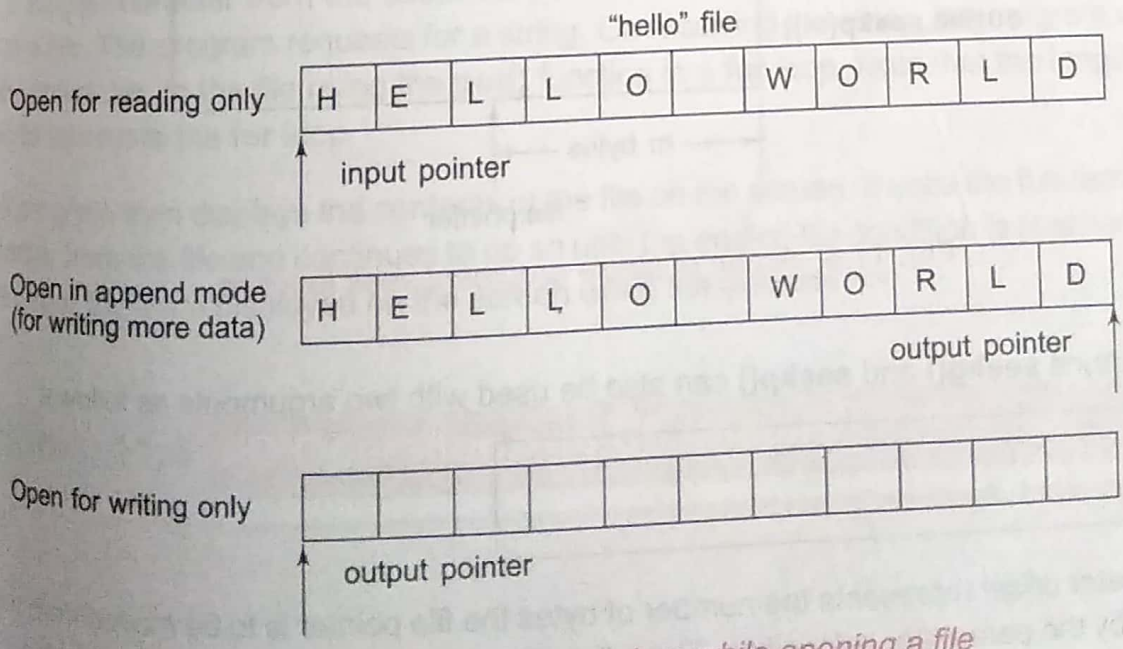

Fig. 11.7 *Action on file pointers while opening a file*

Functions for Manipulation of File Pointers

All the actions on the file pointers as shown in Fig. 11.7 take place automatically by default. How do we then move a file pointer to any other desired position inside the file? This is possible only if we can take control of the movement of the file pointers ourselves. The file stream classes support the following functions to manage such situations:

- **seekg()** Moves get pointer (input) to a specified location.
- **seekp()** Moves put pointer(output) to a specified location.
- **tellg()** Gives the current position of the get pointer.
- **tellp()** Gives the current position of the put pointer.

For example, the statement

```
infile.seekg(10);
```

moves the file pointer to the byte number 10. Remember, the bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing to the 11th byte in the file.

Consider the following statements:

```
ofstream fileout;
fileout.open("hello", ios::app);
int p = fileout.tellp();
```

On execution of these statements, the output pointer is moved to the end of the file "hello" and the value of **p** will represent the number of bytes in the file.

Specifying the offset

We have just seen how to move a file pointer to a desired location using the 'seek' functions. The argument to these functions represents the absolute position in the file. This is shown in Fig. 11.8.

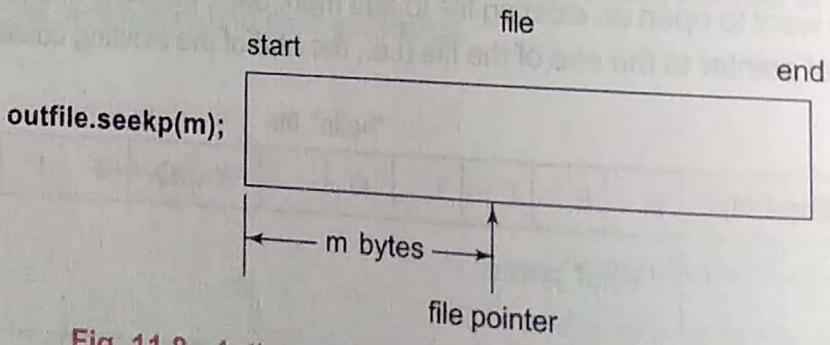

Fig. 11.8 *Action of single argument seek function*

'Seek' functions **seekg()** and **seekp()** can also be used with two arguments as follows:

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

The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the parameter *refposition*. The *refposition* takes one of the following three constants defined in the **ios** class:

- **ios::beg** start of the file
- **ios::cur** current position of the pointer
- **ios::end** End of the file

The **seekg()** function moves the associated file's 'get' pointer while the **seekp()** function moves the associated file's 'put' pointer. Table 11.3 lists some sample pointer offset calls and their actions. fout is an **ofstream** object.

Table 11.3 *Pointer offset calls*

Seek call	*Action*
fin.seekg(o, ios::beg);	Go to start
fin.seekg(o, ios::cur);	Stay at the current position
fin.seekg(o, ios::end);	Go to the end of file
fin.seekg(m,ios::beg);	Move to (m + 1)th byte in the file
fin.seekg(m,ios::cur);	Go forward by m byte form the current position
fin.seekg(-m,ios::cur);	Go backward by m bytes from the current position
fin.seekg(-m,ios::end);	Go backward by m bytes form the end

11.7 SEQUENTIAL INPUT AND OUTPUT OPERATIONS

The file stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, **put()** and **get()**, are designed for handling a single character at a time. Another pair of functions, **write()** and **read()**, are designed to write and read blocks of *binary data*.

put() and get() Functions

The function **put()** writes a single character to the associated stream. Similarly, the function **get()** reads a single character from the associated stream. Program 11.4 illustrates how these functions work on a file. The program requests for a string. On receiving the string, the program writes it, character by character, to the file using the **put()** function in a **for** loop. Note that the length of the string is used to terminate the **for** loop.

The program then displays the contents of the file on the screen. It uses the function **get()** to fetch a character from the file and continues to do so until the end-of-file condition is reached. The character read from the file is displayed on the screen using the operator <<.

Program 11.4 I/O Operations on Characters

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

(Contd.)

```
#include <string.h>

int main()
{
        char string[80];

        cout<<"Enter a string: ";
        cin>>string;

        int len=strlen(string);

                                        //input and output stream
        fstream file;
        cout<<"\nOpening the 'TEXT' file and storing the string in
          it.\n\n";

        file.open("TEXT", ios::in | ios::out);

        for(int i=0;i<len;i++)
          file.put(string[i]);          //put a character to file

                                        //go to the start
        file.seekg(0);

        char ch;
        cout<<"Reading the file contents:  ";
        while(file)
        {
          file.get(ch);                 //get a character from file
          cout<<ch;                     //display it on screen
        }

        return 0;
}
```

The output of Program 11.4 would be:

```
Enter a string: C++_Programming

Opening the 'TEXT' file and storing the string in it.

Reading the file contents:  C++_Programming
```

Note *We have used an* ***fstream*** *object to open the file. Since an* ***fstream*** *object can handle both the input and output simultaneously, we have opened the file in* ***ios::in | ios::out*** *mode. After writing the file, we want to read the entire file and display its contents. Since the file pointer has already moved to the end of the file, we must bring it back to the start of the file. This is done by the statement*

```
file.seekg(0);
```

write() and read() Functions

The functions **write()** and **read()**, unlike the functions **put()** and **get()**, handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in

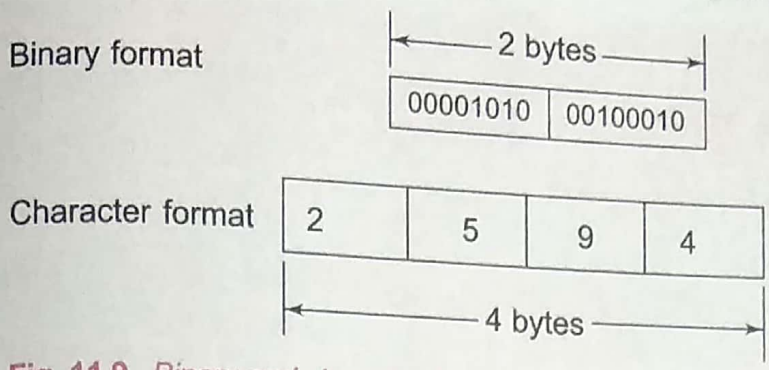

Fig. 11.9 *Binary and character formats of an integer value*

the internal memory. Figure 11.9 shows how an **int** value 2594 is stored in the *binary* and *character* formats. An **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form.

The binary format is more accurate for storing the numbers as they are stored in the exact internal representation. There are no conversions while saving the data and therefore saving is much faster.

The binary input and output functions takes the following form:

```
infile. read ((char *) & V, sizeof (V));
outfile.write ((char *) & V, sizeof (V));
```

These functions take two arguments. The first is the address of the variable **V**, and the second is the length of that variable in bytes. The address of the variable must be cast to type **char** * (i.e., pointer to character type). Program 11.5 illustrates how these two functions are used to save an array of **float** numbers and then recover them for display on the screen.

Program 11.5 I/O Operations on binary files

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

const char * filename = "BINARY";

int main()
{
    float height[4] = {175.5,153.0,167.25,160.70};

    ofstream outfile;
    outfile.open(filename);

    outfile.write((char *) & height, sizeof(height));

    outfile.close();                    // close the file for reading

    for(int i=0; i<4; i++)              // clear array from memory
      height[i] = 0;
    ifstream infile;
    infile.open(filename);
```

(Contd.)

```
        infile.read((char *) & height, sizeof(height));

        for(i=0; i<4; i++)
        {
            cout.setf(ios::showpoint);
            cout << setw(10) << setprecision(2)
                 << height[i];
        }
        infile.close();

        return 0;
}
```

The output of Program 11.5 would be:

```
175.50  153.00  167.25
```

Reading and Writing a Class Object

We mentioned earlier that one of the shortcomings of the I/O system of C is that it cannot handle user-defined data types such as class objects. Since the class objects are the central elements of C++ programming, it is quite natural that the language supports features for writing to and reading from the disk files objects directly. The binary input and output functions **read()** and **write()** are designed to do exactly this job. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For instance, the function **write()** copies a class object from memory byte by byte with no conversion. One important point to remember is that only data members are written to the disk file and the member functions are not.

Program 11.6 illustrates how class objects can be written to and read from the disk files. The length of the object is obtained using the **sizeof** operator. This length represents the sum total of lengths of all data members of the object.

Program 11.6 Reading and Writing Class Objects

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
class INVENTORY
{
        char name[10];              // item name
        int  code;                  // item code
        float cost;                 // cost of each item
  public:
        void readdata(void);
        void writedata(void);
};

void INVENTORY :: readdata(void)    // read from keyboard
```

(Contd.)

```
{
     cout << "Enter name: "; cin >> name;
     cout << "Enter code: "; cin >> code;
     cout << "Enter cost: "; cin >> cost;
}
void INVENTORY :: writedata(void)     // formatted display on
{                                     // screen
     cout << setiosflags(ios::left)
          << setw(10) << name
          << setiosflags(ios::right)
          << setw(10) << code
          << setprecision(2)
          << setw(10) << cost
          << endl;
}
int main()
{
     INVENTORY item[3];               // Declare array of 3 objects

     fstream file;                    // Input and output file

     file.open("STOCK.DAT", ios::in | ios::out);

     cout << "ENTER DETAILS FOR THREE ITEMS \n";
     for(int i=0;i<3;i++)
     {
          item[i].readdata();
          file.write((char *) & item[i],sizeof(item[i]));
     }
     file.seekg(0);                   // reset to start

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

The output of Program 11.6 would be:

```
ENTER DETAILS FOR THREE ITEMS
Enter name:C++
Enter code:101
Enter cost:175
Enter name:FORTRAN
Enter code:102
```

```
Enter cost:150
Enter name:JAVA
Enter code:115
Enter cost:225

OUTPUT

C++             101 175
FORTRAN         102 150
JAVA            115 225
```

The program uses 'for' loop for reading and writing objects. This is possible because we know the exact number of objects in the file. In case, the length of the file is not known, we can determine the file-size in terms of objects with the help of the file pointer functions and use it in the 'for' loop or we may use **while(file)** test approach to decide the end of the file. These techniques are discussed in the next section.

Program 11.7 shows another program that counts the number of objects stored in a file:

Program 11.7 Counting Objects in a File

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

class emp
{
      char name[30];
      int ecode;

  public:
      emp()
      {
      }

      emp(char *n,int c)
      {
      strcpy(name,n);
      ecode=c;
      }
};

void main()
{
      emp e[4];
      e[0]=emp("Amit",1);
      e[1]=emp("Joy",2);
      e[2]=emp("Rahul",3);
      e[3]=emp("Vikas",4);
```

(Contd.)

```
    fstream file;
    file.open("Employee.dat",ios::in | ios::out);

    int i;
    for(i=0;i<4;i++)
    file.write((char *) &e[i],sizeof(e[i]));
    file.seekg(0,ios::end);
    int end=file.tellg();

    cout<<"Number of objects stored in Employee.dat is:"<<end/
    sizeof(emp);
}
```

The output of Program 11.7 would be:

```
Number of objects stored in Employee.dat is: 4
```

The above program uses the **tellg()** function to determine the current position of the get pointer, which in turn specifies the length of the file in bytes.

11.8 UPDATING A FILE: RANDOM ACCESS

Updating is a routine task in the maintenance of any data file. The updating would include one or more of the following tasks:

- Displaying the contents of a file
- Modifying an existing item
- Adding a new item
- Deleting an existing item

These actions require the file pointers to move to a particular location that corresponds to the item/object under consideration. This can be easily implemented if the file contains a collection of items/objects of equal lengths. In such cases, the size of each object can be obtained using the statement

```
int object_length = sizeof(object);
```

Then, the location of a desired object, say the mth object, may be obtained as follows:

```
int location = m * object_length;
```

The **location** gives the byte number of the first byte of the mth object. Now, we can set the file pointer to reach this byte with the help of **seekg()** or **seekp()**.

We can also find out the total number of objects in a file using the **object_length** as follows:

```
int n = file_size/object_length;
```

The file_size can be obtained using the function **tellg()** or **tellp()** when the file pointer is located at the end of the file.

Program 11.8 illustrates how some of the tasks described above are carried out. The program uses the "STOCK.DAT" file created using Program 11.6 for five items and performs the following operations on the file:

1. Adds a new item to the file
2. Modifies the details of an item
3. Displays the contents of the file

Program 11.8 File Updating :: Random Access

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

class INVENTORY

{
      char name[10];
      int code;
      float cost;
  public:
      void getdata(void)
      {
          cout << "Enter name: "; cin >> name;
          cout << "Enter code: "; cin >> code;
          cout << "Enter cost: "; cin >> cost;
      }
      void putdata(void)
      {
          cout << setw(10) << name
               << setw(10) << code
               << setprecision(2) << setw(10) << cost
               << endl;
      }
};   // End of class definition
int main()
{
      INVENTORY item;
      fstream inoutfile;                    // input/output stream

        inoutfile.open("STOCK.DAT", ios:: ate | ios:: in |
                        ios::out | ios::binary);
         inoutfile.seekg(0,ios::beg);       // go to start

         cout << "CURRENT CONTENTS OF STOCK" << "\n";

         while(inoutfile.read((char *) & item, sizeof item))
         {
              item.putdata();
```

```
    }
    inoutfile.clear();                    // turn of EOF flag

/* >>>>>>>>>>>>> Add one more item <<<<<<<<<<<<<< */

cout << "\nADD AN ITEM\n";
item.getdata();
char ch;
cin.get(ch);
inoutfile.write((char *) & item, sizeof item);

// Display the appended file
inoutfile.seekg(0);                       // go to the start

cout << "CONTENTS OF APPENDED FILE \n";

while(inoutfile.read((char *) & item, sizeof item))
{
      item.putdata();
}

// Find number of objects in the file
int last = inoutfile.tellg();
int n = last/sizeof(item);

cout << "Number of objects = " << n << "\n";
cout << "Total bytes in the file = " << last << "\n";

/* >>>>>>>> MODIFY THE DETAILS OF AN ITEM <<<<<<<<< */

cout << "Enter object number to be updated \n";
int object;
cin >> object;

cin.get(ch);

int location = (object-1) * sizeof(item);

if(inoutfile.eof())
inoutfile.clear();

inoutfile.seekp(location);
cout << "Enter new values of the object \n";
item.getdata();
cin.get(ch);

inoutfile.write((char *) & item, sizeof item) << flush;

/* >>>>>>>>>>>>> SHOW UPDATED FILE <<<<<<<<<<<<<< */

inoutfile.seekg(0); //go to the start
```

```
        cout << "CONTENTS OF UPDATED FILE \n";

        while(inoutfile.read((char *) & item, sizeof item))
        {
              item.putdata();
        }
        inoutfile.close();

      return 0;
} // End of main
```

The output of Program 11.8 would be:

```
CURRENT CONTENTS OF STOCK
AA        11        100
BB        22        200
CC        33        300
DD        44        400
XX        99        900

ADD AN ITEM
Enter name: YY
Enter code: 10
Enter cost: 101

CONTENTS OF APPENDED FILE
AA        11        100
BB        22        200
CC        33        300
DD        44        400
XX        99        900
YY        10        101

Number of objects = 6
Total bytes in the files = 96
Enter object number to be updated
6
Enter new values of the object
Enter name: ZZ
Enter code: 20
Enter cost: 201

CONTENTS OF UPDATED FILE
AA        11        100
BB        22        200
CC        33        300
DD        44        400
XX        99        900
ZZ        20        201
```

We are using the **fstream** class to declare the file streams. The **fstream** class inherits two buffers, one for input and another for output, and synchronizes the movement of the file pointers on these buffers. That is, whenever we read from or write to the file, both the pointers move in tandem. Therefore, at any point of time, both the pointers point to the same byte.

Since we have to add new objects to the file as well as modify some of the existing objects, we open the file using **ios::ate** option for input and output operations. Remember, the option **ios::app** allows us to add data to the end of the file only. The **ios::ate** mode sets the file pointers at the end of the file when opening it. We must therefore move the 'get' pointer to the beginning of the file using the function **seekg()** to read the existing contents of the file.

At the end of reading the current contents of the file, the program sets the EOF flag on. This prevents any further reading from or writing to the file. The EOF flag is turned off by using the function **clear()**, which allows access to the file once again.

After appending a new item, the program displays the contents of the appended file and also the total number of objects in the file and the memory space occupied by them.

To modify an object, we should reach to the first byte of that object. This is achieved using the statements

```
int location = (object-1) * sizeof(item);
inoutfile.seekp(location);
```

The program accepts the number and the new values of the object to be modified and updates it. Finally, the contents of the appended and modified file are displayed.

Remember, we are opening an existing file for reading and updating the values. It is, therefore, essential that the data members are of the same type and declared in the same order as in the existing file. Since, the member functions are not stored, they can be different.

11.9 ERROR HANDLING DURING FILE OPERATIONS

So far we have been opening and using the files for reading and writing on the assumption that everything is fine with the files. This may not be true always. For instance, one of the following things may happen when dealing with the files:

1. A file which we are attempting to open for reading does not exist.
2. The file name used for a new file may already exist.
3. We may attempt an invalid operation such as reading past the end-of-file.
4. There may not be any space in the disk for storing more data.
5. We may use an invalid file name.
6. We may attempt to perform an operation when the file is not opened for that purpose.

The C++ file stream inherits a 'stream-state' member from the class **ios**. This member records information on the status of a file that is being currently used. The stream state member uses bit fields to store the status of the error conditions stated above.

The class **ios** supports several member functions that can be used to read the status recorded in a file stream. These functions along with their meanings are listed in Table 11.4.

Table 11.4 Error handling functions

Function	Return value and meaning
eof()	Returns *true* (non-zero value) if end-of-file is encountered while reading; Otherwise returns *false(zero)*
fail()	Returns *true* when an input or output operation has failed
bad()	Returns *true* if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is *false*, it may be possible to recover from any other error reported, and continue operation.
good()	Returns true if no error has occurred. This means, all the above functions are false. For instance, if **file.good()** is *true*, all is well with the stream **file** and we can proceed to perform I/O operations. When it returns *false*, no further operations can be carried out.

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

```
.....
.....
ifstream infile;
infile.open("ABC");
while(!infile.fail())
{
.....
.....   (process the file)
.....
}
if(infile.eof())
{
.....   (terminate program normally)
}
else
        if(infile.bad() )
{
.....(report fatal error)
}
else
{
infile.clear();
.....
.....                          // clear error state
}
.....
.....
```

The function **clear()** (which we used in the previous section as well) resets the error state so that further operations can be attempted.

Remember that we have already used statements such as

```
while(infile)
{
    .....
    .....
}
```

and

```
while(infile.read(....))
{
    .....
    .....
}
```

Here, **infile** becomes *false* (zero) when end of the file is reached (and **eof()** becomes *true*).

11.10 COMMAND-LINE ARGUMENTS

Like C, C++ too supports a feature that facilitates the supply of arguments to the **main()** function. These arguments are supplied at the time of invoking the program. They are typically used to pass the names of data files. Example:

```
C > exam data results
```

Here, *exam* is the name of the file containing the program to be executed, and data and results are the filenames passed to the program as *command-line* arguments.

The command-line arguments are typed by the user and are delimited by a space. The first argument is always the filename (command name) and contains the program to be executed. But, how do these arguments get into the program?

The **main()** functions which we have been using up to now without any arguments can take two arguments as shown below:

```
main(int argc, char * argv[])
```

The first argument **argc** (known as *argument counter*) represents the number of arguments in the command line. The second argument **argv** (known as *argument vector*) is an array of **char** type pointers that points to the command-line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line

```
C > exam data results
```

the value of **argc** would be 3 and the **argv** would be an array of three pointers to strings as shown below:

```
argv[0] ---> exam
argv[1] ---> data
argv[2] ---> results
```

Note that **argv[0]** always represents the command name that invokes the program. The character pointers **argv[1]** and **argv[2]** can be used as file names in the file opening statements as shown below:

```
.....
.....                                    // open data file for reading
infile.open(argv[1]);
.....
.....                                    // open results file for
outfile.open(argv[2]);                       // writing

.....
.....
```

Program 11.9 illustrates the use of the command-line arguments for supplying the file names. The command line is

```
test ODD EVEN
```

The program creates two files called **ODD** and **EVEN** using the command-line arguments, and a set of numbers stored in an array are written to these files. Note that the odd numbers are written to the file **ODD** and the even numbers are written to the file **EVEN**. The program then displays the contents of the files.

Program 11.9 Command-Line Arguments

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

int main(int argc, char * argv[])
{
        int number[9] = {11,22,33,44,55,66,77,88,99};

        if(argc != 3)
        {
          cout << "argc = " << argc << "\n";
            cout << "Error in arguments \n";
            exit(1);
        }

        ofstream fout1, fout2;

        fout1.open(argv[1]);

        if(fout1.fail())
        {
            cout << "could not open the file"
                 << argv[1] << "\n";
            exit(1);
        }
        fout2.open(argv[2]);
```

```
        if(fout2.fail())
        {
            cout << "could not open the file "
                 << argv[2] << "\n";
            exit(1);
        }
        for(int i=0; i<9; i++)
        {
            if(number[i] % 2 == 0)
            fout2 << number[i] << " ";  // write to EVEN file
            else
            fout1 << number[i] << " ";  // write to ODD file
        }
        fout1.close();
        fout2.close();

        ifstream fin;
        char ch;
        for(i=1; i<argc; i++)
        {
            fin.open(argv[i]);
            cout << "Contents of " << argv[i] << "\n";
            do
            {
                fin.get(ch);            // read a value
                cout << ch;             // display it
            }
            while(fin);
            cout << "\n\n";
            fin.close();
        }
return 0;
}
```

The output of Program 11.9 would be:

```
Contents of ODD
11 33 55 77 99

Contents of EVEN
22 44 66 88
```

SUMMARY

- The C++ I/O system contains classes such as **ifstream, ofstream** and **fstream** to deal with file handling. These classes are derived from **fstreambase** class and are declared in a header file *iostream*.
- A file can be opened in two ways by using the constructor function of the class and using the member function **open()** of the class.

(Contd.)