Every program takes some data as input and generates processed data as output following the familiar input-process-output cycle. It is, therefore, essential to know how to provide the input data and how to present the results in a desired form. We have, in the earlier chapters, used cin and cout with the operators >> and << for the input and output operations. But we have not so far discussed as to how to control the way the output is printed. C++ supports a rich set of I/O functions and operations to do this. Since these functions use the advanced features of C++ (such as classes, derived classes and virtual functions), we need to know a lot about them before really implementing the C++ I/O operations.

Remember, C++ supports all of C's rich set of I/O functions. We can use any of them in the C++ programs. But we restrained from using them due to two reasons. First, I/O methods in C++ support the concepts of OOP and secondly, I/O methods in C cannot handle the user-defined data types such as class objects.
C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files. We will discuss in this chapter, how stream classes support the console-oriented input-output operations. File-oriented I/O operations will be discussed in the next chapter.

10.2 C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as stream.

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream. In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream as illustrated in Fig. 10.1.

![Fig. 10.1 Data streams](image)

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device. Therefore, a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include cin and cout which have been used very often in our earlier programs. We know that cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.
10.3 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Figure 10.2 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. This file should be included in all the programs that communicate with the console unit.

![Diagram showing the hierarchy of stream classes]

As seen in the Fig. 10.2, ios is the base class for istream (input stream) and ostream (output stream) which are, in turn, base classes for iostream (input/output stream). The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the class ostream (through inheritance) provides the facilities for formatted output. The class iostream provides the facilities for handling both input and output streams. Three classes, namely, istream_withassign, ostream_withassign, and iostream_withassign add assignment operators to these classes. Table 10.1 gives the details of these classes.

10.4 Unformatted I/O Operations

Overloaded Operators >> and <<

We have used the objects cin and cout (pre-defined in the iostream file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >> operator is overloaded in the
### Table 10.1  Stream classes for console operations

<table>
<thead>
<tr>
<th>Class name</th>
<th>Contents</th>
</tr>
</thead>
</table>
| ios (General input/output stream class) | - Contains basic facilities that are used by all other input and output classes  
- Also contains a pointer to a buffer object (**streambuf** object)  
- Declares constants and functions that are necessary for handling formatted input and output operations |
| istream (input stream) | - Inherits the properties of **ios**  
- Declares input functions such as **get()**, **getline()** and **read()**  
- Contains overloaded extraction operator >> |
| ostream (output stream) | - Inherits the properties of **ios**  
- Declares output functions **put()** and **write()**  
- Contains overloaded insertion operator << |
| iostream (input/output stream) | - Inherits the properties of **ios istream** and **ostream** through multiple inheritance and thus contains all the input and output functions |
| streambuf | - Provides an interface to physical devices through buffers  
- Acts as a base for **filebuf** class used for **ios** files |

**istream** class and **<<** is overloaded in the **ostream** class. The following is the general format for reading data from the keyboard:

```cpp
    cin >> variable1 >> variable2 >> .... >> variableN
```

`variable1, variable2, ...` are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be:

```plaintext
data1 data2 ....... dataN
```

The input data are separated by white spaces and should match the type of variable in the **cin** list. Spaces, newlines and tabs will be skipped.

The operator **>>** reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example, consider the following code:

```cpp
    int code;  
    cin >> code;
```

Suppose the following data is given as input:

```plaintext
42580
```
The operator will read the characters upto 8 and the value 4258 is assigned to code. The character D remains in the input stream and will be input to the next cin statement. The general form for displaying data on the screen is:

```cpp
    cout << item1 << item2 << .... << itemN
```

The items item1 through itemN may be variables or constants of any basic type. We have used such statements in a number of examples illustrated in previous chapters.

**put() and get() Functions**

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char *) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char *) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object.

Example:

```cpp
    char c;
    cin.get(c);  // get a character from keyboard
    cin.get(c);  // and assign it to c
    while(c != '\n')
    {
        cin.get(c);  // display the character on screen
        cout << c;   // get another character
    }
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character. The above **while** loop will not work properly if the statement

```cpp
    cin >> c;
```

is used in place of

```cpp
    cin.get(c);
```

**note**

Try using both of them and compare the results.
The `get(void)` version is used as follows:

```cpp
.....
char c;
c = cin.get();  // cin.get(c); replaced
.....
.....
```

The value returned by the function `get()` is assigned to the variable `c`.

The function `put()`, a member of `ostream` class, can be used to output a line of text, character by character. For example,

```cpp
cout.put('x');
```

displays the character `x` and

```cpp
cout.put(ch);
```

displays the value of variable `ch`.

The variable `ch` must contain a character value. We can also use a number as an argument to the function `put()`. For example,

```cpp
cout.put(68);
```

displays the character D. This statement will convert the `int` value 68 to a `char` value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```cpp
char c;
cin.get(c);  // read a character

while(c != '\n')
{
    cout.put(c);  // display the character on screen
cin.get(c);
}
```

Program 10.1 illustrates the use of these two character handling functions.
#include <iostream>
using namespace std;

int main()
{
    int count = 0;
    char c;

    cout << "INPUT TEXT\n";
    cin.get(c);

    while(c != '\n')
    {
        cout.put(c);
        count++;
        cin.get(c);
    }

    cout << "\nNumber of characters = " << count << "\n";
    return 0;
}

Input
Object Oriented Programming

Output
Object Oriented Programming
Number of characters = 27

When we type a line of input, the text is sent to the program as soon as we press the RETURN key. The program then reads one character at a time using the statement cin.get(c); and displays it using the statement cout.put(c). The process is terminated when the newline character is encountered.

**note**

getline() and write() Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character (transmitted by the RETURN key). This function can be invoked by using the object cin as follows:
This function call invokes the function getline() which reads character input into the variable line. The reading is terminated as soon as either the newline character ‘\n’ is encountered or size-1 characters are read (whichever occurs first). The newline character is read but not saved. Instead, it is replaced by the null character. For example, consider the following code:

```cpp
char name[20];
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

Bjarne Stroustrup <press RETURN>

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

Object Oriented Programming <press RETURN>

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

Remember, the two blank spaces contained in the string are also taken into account.

We can also read strings using the operator >> as follows:

```cpp
cin >> name;
```

But remember cin can read strings that do not contain white spaces. This means that cin can read just one word and not a series of words such as “Bjarne Stroustrup”. But it can read the following string correctly:

Bjarne_Stroustrup

After reading the string, cin automatically adds the terminating null character to the character array.

Program 10.2 demonstrates the use of >> and getline() for reading the strings.
```c
int main()
{
    int size = 20;
    char city[20];

    cout << "Enter city name: \n";
    cin >> city;
    cout << "City name:" << city << "\n\n";

    cout << "Enter city name again: \n";
    cin.getline(city, size);
    cout << "City name now: " << city << "\n\n";

    cout << "Enter another city name: \n";
    cin.getline(city, size);
    cout << "New city name: " << city << "\n\n";

    return 0;
}
```

The output of Program 10.2 would be:

**First run**

Enter city name:
Delhi
City name: Delhi

Enter city name again:
City name now:
Enter another city name:
Chennai
New city name: Chennai

**Second run**

Enter city name:
New Delhi
City name: New

Enter city name again:
City name now: Delhi

Enter another city name:
Greater Bombay
New city name: Greater Bombay
During first run, the newline character '\n' at the end of "Delhi" which is waiting in the input queue is read by the getline() that follows immediately and therefore it does not wait for any response to the prompt 'Enter city name again:'. The character '\n' is read as an empty line. During the second run, the word "Delhi" (that was not read by cin) is read by the function getline() and, therefore, here again it does not wait for any input to the prompt 'Enter city name again:'. Note that the line of text "Greater Bombay" is correctly read by the second cin.getline(city,size); statement.

The write() function displays an entire line and has the following form:

```cpp
cout.write(line, size)
```

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display. Note that it does not stop displaying the characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line. Program 10.3 illustrates how write() method displays a string.

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

using namespace std;

int main()
{
    char * string1 = "C++ ";
    char * string2 = "Programming";
    int m = strlen(string1);
    int n = strlen(string2);

    for(int i=1; i<n; i++)
    {
        cout.write(string2,i);
        cout << "\n";
    }

    for(i=n; i>0; i--)
    {
        cout.write(string2,i);
        cout << "\n";
    }
}

(Contd)
// concatenating strings
cout.write(string1,m).write(string2,n);
cout << "\n";
// crossing the boundary
cout.write(string1,10);
return 0;
}

Look at the output of Program 10.3:

P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++ Progr

The last line of the output indicates that the statement

cout.write(string1, 10);

displays more characters than what is contained in string1.

It is possible to concatenate two strings using the write() function. The statement

cout.write(string1, m).write(string2, n);
is equivalent to the following two statements:

    cout.write(string1, m);
    cout.write(string2, n);

### 10.5 Formatted Console I/O Operations

C++ supports a number of features that could be used for formatting the output. These features include:

- **ios class functions and flags.**
- **Manipulators.**
- **User-defined output functions.**

The **ios** class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are listed in Table 10.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width()</td>
<td>To specify the required field size for displaying an output value</td>
</tr>
<tr>
<td>precision()</td>
<td>To specify the number of digits to be displayed after the decimal point of a float value</td>
</tr>
<tr>
<td>fill()</td>
<td>To specify a character that is used to fill the unused portion of a field</td>
</tr>
<tr>
<td>setf()</td>
<td>To specify format flags that can control the form of output display (such as left-justification and right-justification)</td>
</tr>
<tr>
<td>unsetf()</td>
<td>To clear the flags specified</td>
</tr>
</tbody>
</table>

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Table 10.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

<table>
<thead>
<tr>
<th>Manipulators</th>
<th>Equivalent ios function</th>
</tr>
</thead>
<tbody>
<tr>
<td>setw()</td>
<td>width()</td>
</tr>
<tr>
<td>setprecision()</td>
<td>precision()</td>
</tr>
<tr>
<td>setfill()</td>
<td>fill()</td>
</tr>
<tr>
<td>setiosflags()</td>
<td>setf()</td>
</tr>
<tr>
<td>resetiosflags()</td>
<td>unsetf()</td>
</tr>
</tbody>
</table>
In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats. The following sections will provide details of how to use the pre-defined formatting functions and how to create new ones.

**Defining Field Width: width()**

We can use the `width()` function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it, as shown below:

```cpp
cout.width(w);
```

where `w` is the field width (number of columns). The output will be printed in a field of `w` characters wide at the right end of the field. The `width()` function can specify the field width for only one item (the item that follows immediately). After printing one item (as per the specifications) it will revert back to the default. For example, the statements

```cpp
cout.width(5);
cout << 543 << 12 << "\n";
```

will produce the following output:

```
   5  4  3  1  2
```

The value 543 is printed right-justified in the first five columns. The specification `width(5)` does not retain the setting for printing the number 12. This can be improved as follows:

```cpp
cout.width(5);
cout << 543;
cout.width(5);
cout << 12 << "\n";
```

This produces the following output:

```
   5  4  3       1  2
```

Remember that the field width should be specified for each item separately. C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value. Program 10.4 demonstrates how the function `width()` works.
#include <iostream>
using namespace std;

int main()
{
    int items[4] = {10, 8, 12, 15};
    int cost[4] = {75, 100, 60, 99};

    cout.width(5);
    cout << "ITEMS";
    cout.width(8);
    cout << "COST";

    cout.width(15);
    cout << "TOTAL VALUE" << "\n";

    int sum = 0;

    for(int i=0; i<4; i++)
    {
        cout.width(5);
        cout << items[i];

        cout.width(8);
        cout << cost[i];

        int value = items[i] * cost[i];
        cout.width(15);
        cout << value << "\n";
        sum = sum + value;
    }
    cout << "\nGrand Total = ";

    cout.width(2);
    cout << sum << "\n";

    return 0;
}
The output of Program 10.4 would be:

<table>
<thead>
<tr>
<th>ITEMS</th>
<th>COST</th>
<th>TOTAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>75</td>
<td>750</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>800</td>
</tr>
<tr>
<td>12</td>
<td>60</td>
<td>720</td>
</tr>
<tr>
<td>15</td>
<td>99</td>
<td>1485</td>
</tr>
</tbody>
</table>

Grand Total = 3755

**note**

A field of width two has been used for printing the value of sum and the result is not truncated. A good gesture of C++!

**Setting Precision: `precision()`**

By default, the floating numbers are printed with six digits after the decimal point. However, we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the `precision()` member function as follows:

```cpp
cout.precision(d);
```

where d is the number of digits to the right of the decimal point. For example, the statements

```cpp
cout.precision(3);
cout << sqrt(2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";
```

will produce the following output:

1.141 (truncated)
3.142 (rounded to the nearest cent)
2.5  (no trailing zeros)

Not that, unlike the function `width()`, `precision()` retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

We can set different values to different precision as follows:

```cpp
cout.precision(3);
```
cout << sqrt(2) << "\n";'  
cout.precision(5);       // Reset the precision
cout << 3.14159 << "\n";

We can also combine the field specification with the precision setting. Example:

cout.precision(2);
cout.width(5);
cout << 1.2345;

The first two statements instruct: ‘print two digits after the decimal point in a field of five character width’. Thus, the output will be:

```
1 2 3
```

Program 10.5 shows how the functions `width()` and `precision()` are jointly used to control the output format.

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    cout << "Precision set to 3 digits \n\n"
      << std::precision(3);

    cout.width(10);
    cout << "VALUE"
      << std::width(15);
    cout << "SQRRT_OF_VALUE" << "\n"
      << std::width(13);
    cout << sqrt(n) << "\n"
      << std::width(13);
}
```

(Contd)
cout << "\n Precision set to 5 digits \n\n";
cout.precision(5); // precision parameter changed
cout << " sqrt(10) = " << sqrt(10) << "\n\n";
cout.precision(0); // precision set to default
cout << " sqrt(10) = " << sqrt(10) << " (default setting)\n";

return 0;

Here is the output of Program 10.5

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SQRT_OF_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.41</td>
</tr>
<tr>
<td>3</td>
<td>1.73</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2.24</td>
</tr>
</tbody>
</table>

Precision set to 5 digits

sqrt(10) = 3.1623

sqrt(10) = 3.162278 (default setting)

--- note ---

Observe the following from the output:

1. The output is rounded to the nearest cent (i.e., 1.6666 will be 1.67 for two digit precision but 1.3333 will be 1.33).
2. Trailing zeros are truncated.
3. Precision setting stays in effect until it is reset.
4. Default precision is 6 digits.

--- Filling and Padding: fill() ---

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:
cout.fill(ch);

Where ch represents the character which is used for filling the unused positions. Example:

cout.fill('*');
cout.width(10);
cout << 5250 << "\n";

The output would be:

```
* * * * * 5 2 5 0
```

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily. Like precision(), fill() stays in effect till we change it. See Program 10.6 and its output.

```
PADDING WITH fill()
#include <iostream>

using namespace std;

int main( )
{
    cout.fill('<');

    cout.precision(3);
    for(int n=1; n<=6; n++)
    {
        cout.width(5);
        cout << n;
        cout.width(10);
        cout << 1.0 / float(n) << "\n";
        if (n == 3)
            cout.fill('>');
    }
    cout << "\nPadding changed \n\n";
    cout.fill('#');  // fill( ) reset
    cout.width (15);
    cout << 12.345678 << "\n";

    return 0;
}
```

PROGRAM 10.6
The output of Program 10.6 would be:

```
<<<<1<<<<<<<<<<1
<<<<2<<<<<<<<0.5
<<<<3<<<<0.333
>>>>4>>>>0.25
>>>>5>>>>0.2
>>>>6>>0.167

Padding changed

########12.346
```

**Formatting Flags, Bit-fields and setf()**

We have seen that when the function `width()` is used, the value (whether text or number) is printed right-justified in the field width created. But, it is a usual practice to print the text left-justified. How do we get a value printed left-justified? Or, how do we get a floating-point number printed in the scientific notation?

The `setf()`, a member function of the `ios` class, can provide answers to these and many other formatting questions. The `setf()` (setf stands for set flags) function can be used as follows:

```
cout.setf(arg1, arg2)
```

The `arg1` is one of the formatting flags defined in the class `ios`. The formatting flag specifies the format action required for the output. Another `ios` constant, `arg2`, known as bit field specifies the group to which the formatting flag belongs.

Table 10.4 shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive. Examples:

```
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
```

Note that the first argument should be one of the group members of the second argument.

Consider the following segment of code:

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout << "TABLE 1" << "\n";
```
Table 10.4  Flags and bit fields for setf() function

<table>
<thead>
<tr>
<th>Format required</th>
<th>Flag (arg1)</th>
<th>Bit-field (arg2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-justified output</td>
<td>ios :: left</td>
<td>ios :: adjustfield</td>
</tr>
<tr>
<td>Right-justified output</td>
<td>ios :: right</td>
<td>ios :: adjustfield</td>
</tr>
<tr>
<td>Padding after sign or base</td>
<td>ios :: internal</td>
<td>ios :: adjustfield</td>
</tr>
<tr>
<td>Indicator (like +##20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scientific notation</td>
<td>ios :: scientific</td>
<td>ios :: floatfield</td>
</tr>
<tr>
<td>Fixed point notation</td>
<td>ios :: fixed</td>
<td>ios :: floatfield</td>
</tr>
<tr>
<td>Decimal base</td>
<td>ios :: dec</td>
<td>ios :: basefield</td>
</tr>
<tr>
<td>Octal base</td>
<td>ios :: oct</td>
<td>ios :: basefield</td>
</tr>
<tr>
<td>Hexadecimal base</td>
<td>ios :: hex</td>
<td>ios :: basefield</td>
</tr>
</tbody>
</table>

This will produce the following output:

```
TABLE 1 * * * * * * *
```

The statements

```
cout.fill('*');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);

cout << -12.34567 << "\n";
```

will produce the following output:

```
- * * * * * 1 . 2 3 5 e + 0 1
```

note

The sign is left-justified and the value is right left-justified. The space between them is padded with stars. The value is printed accurate to three decimal places in the scientific notation.

Displaying Trailing Zeros and Plus Sign

If we print the numbers 10.75, 25.00 and 15.50 using a field width of, say, eight positions, with two digits precision, then the output will be as follows:

```
1 0 . 7 5
2 5
1 5 . 5
```
Note that the trailing zeros in the second and third items have been truncated.

Certain situations, such as a list of prices of items or the salary statement of employees, require trailing zeros to be shown. The above output would look better if they are printed as follows:

10.75
25.00
15.50

The `setf()` can be used with the flag `ios::showpoint` as a single argument to achieve this form of output. For example,

```cpp
cout.setf(ios::showpoint); // display trailing zeros
```

would cause `cout` to display trailing zeros and trailing decimal point. Under default precision, the value 3.25 will be displayed as 3.250000. Remember, the default precision assumes a precision of six digits.

Similarly, a plus sign can be printed before a positive number using the following statement:

```cpp
cout.setf(ios::showpos); // show +sign
```

For example, the statements

```cpp
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::internal, ios::adjustfield);
cout.width(10);
cout << 275.5 << "\n";
```

will produce the following output:

```
+   275.500
```

The flags such as `showpoint` and `showpos` do not have any bit fields and therefore are used as single arguments in `setf()`. This is possible because the `setf()` has been declared as an overloaded function in the class `ios`. Table 10.5 lists the flags that do not possess a named bit field. These flags are not mutually exclusive and therefore can be set or cleared independently.
Table 10.5  Flags that do not have bit fields

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios :: showbase</td>
<td>Use base indicator on output</td>
</tr>
<tr>
<td>ios :: showpos</td>
<td>Print + before positive numbers</td>
</tr>
<tr>
<td>ios :: showpoint</td>
<td>Show trailing decimal point and zeroes</td>
</tr>
<tr>
<td>ios :: uppercase</td>
<td>Use uppercase letters for hex output</td>
</tr>
<tr>
<td>ios :: skipus</td>
<td>Skip white space on input</td>
</tr>
<tr>
<td>ios :: unitbuf</td>
<td>Flush all streams after insertion</td>
</tr>
<tr>
<td>ios :: stdio</td>
<td>Flush stdout and stderr after insertion</td>
</tr>
</tbody>
</table>

Program 10.7 demonstrates the setting of various formatting flags using the overloaded setf() function.

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    cout.fill('*');
    cout.setf(ios::left, ios::adjustfield);
    cout.width(10);
    cout << "VALUE";

    cout.setf(ios::right, ios::adjustfield);
    cout.width(15);
    cout << "SQRT OF VALUE" << "\n";

    cout.fill('.');
    cout.precision(4);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.setf(ios::fixed, ios::floatfield);

    for(int n=1; n<=10; n++)
    {
        cout.setf(ios::internal, ios::adjustfield);
        cout.width(5);
        cout << n;

        cout.setf(ios::right, ios::adjustfield);
        cout.width(20);
        cout << sqrt(n) << "\n";
    }

    (Contd)
```
The output of Program 10.7 would be:

```
VALUE********SQR T OF VALUE
+...1.............+1.0000
+...2.............+1.4142
+...3.............+1.7321
+...4.............+2.0000
+...5.............+2.2361
+...6.............+2.4495
+...7.............+2.6458
+...8.............+2.8284
+...9.............+3.0000
+..10.............+3.1623
```

\[ \text{SQR T(100)} = +1.0000e+001 \]

**note**

1. The flags set by `setf()` remain effective until they are reset or unset.
2. A format flag can be reset any number of times in a program.
3. We can apply more than one format control jointly on an output value.
4. The `setf()` sets the specified flags and leaves others unchanged.

### 10.6 Managing Output with Manipulators

The header file `iomanip` provides a set of functions called *manipulators* which can be used to manipulate the output formats. They provide the same features as that of the `ios` member functions and flags. Some manipulators are more convenient to use than their counterparts in the class `ios`. For example, two or more manipulators can be used as a chain in one statement as shown below:

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

This kind of concatenation is useful when we want to display several columns of output.
The most commonly used manipulators are shown in Table 10.6. The table also gives their meaning and equivalents. To access these manipulators, we must include the file iomanip in the program.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Meaning</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>setw (int w)</td>
<td>Set the field width to w.</td>
<td>width( )</td>
</tr>
<tr>
<td>setprecision(int d)</td>
<td>Set the floating point precision to d.</td>
<td>precision( )</td>
</tr>
<tr>
<td>setfill(int c)</td>
<td>Set the fill character to c.</td>
<td>fill( )</td>
</tr>
<tr>
<td>setiosflags(long f)</td>
<td>Set the format flag f.</td>
<td>setf( )</td>
</tr>
<tr>
<td>resetiosflags(long f)</td>
<td>Clear the flag specified by f.</td>
<td>unsetf( )</td>
</tr>
<tr>
<td>endl</td>
<td>Insert new line and flush stream.</td>
<td>&quot;\n&quot;</td>
</tr>
</tbody>
</table>

Some examples of manipulators are given below:

```cpp
cout << setw(10) << 12345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters. The output can be made left-justified by modifying the statement as follows:

```cpp
cout << setw(10) << setiosflags(ios::left) << 12345;
```

One statement can be used to format output for two or more values. For example, the statement

```cpp
cout << setw(5) << setprecision(2) << 1.2345
    << setw(10) << setprecision(4) << sqrt(2)
    << setw(15) << setiosflags(ios::scientific) << sqrt(3);
<< endl;
```

will print all the three values in one line with the field sizes of 5, 10, and 15 respectively. Note that each output is controlled by different sets of format specifications.

We can jointly use the manipulators and the ios functions in a program. The following segment of code is valid:

```cpp
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout << setprecision(4);
cout << setiosflags(ios::scientific);
cout << setw(10) << 123.45678;
```
There is a major difference in the way the manipulators are implemented as compared to the \texttt{ios} member functions. The \texttt{ios} member function return the previous format state which can be used later, if necessary. But the manipulator does not return the previous format state. In case, we need to save the old format states, we must use the \texttt{ios} member functions rather than the manipulators. Example:

\begin{verbatim}
cout.precision(2); // previous state
int p = cout.precision(4); // current state;
\end{verbatim}

When these statements are executed, \texttt{p} will hold the value of 2 (previous state) and the new format state will be 4. We can restore the previous format state as follows:

\begin{verbatim}
cout.precision(p); // p = 2
\end{verbatim}

Program 10.8 illustrates the formatting of the output values using both manipulators and \texttt{ios} functions.

\begin{verbatim}
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout.setf(ios::showpoint);

    cout << setw(5) << "n"
        << setw(15) << "Inverse of n"
        << setw(15) << "Sum of terms\n\n";

    double term, sum = 0;

    for(int n=1; n<=10; n++)
    {
        term = 1.0 / float(n);
        sum = sum + term;
        cout << setw(5) << n
             << setw(14) << setprecision(4)
\end{verbatim}

(Contd)
The statement

    cout << 36 << unit;

will produce the following output

    36 inches

We can also create manipulators that could represent a sequence of operations. Example:

    ostream & show(ostream & output)
    {
        output.setf(ios::showpoint);
        output.setf(ios::showpos);
        output << setw(10);
        return output;
    }

This function defines a manipulator called show that turns on the flags showpoint and showpos declared in the class ios and sets the field width to 10.

Program 10.9 illustrates the creation and use of the user-defined manipulators. The program creates two manipulators called currency and form which are used in the main program.

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

// user-defined manipulators
ostream & currency(ostream & output)
{
    output << "Rs";
    return output;
}

ostream & form(ostream & output)
{
    output.setf(ios::showpos);
    output.setf(ios::showpoint);
}
```

(Contd)
The output of Program 10.9 would be:

Rs**+7864.50

Note that form represents a complex set of format functions and manipulators.

**SUMMARY**

- In C++, the I/O system is designed to work with different I/O devices. This I/O system supplies an interface called ‘stream’ to the programmer, which is independent of the actual device being used.
- A stream is a sequence of bytes and serves as a source or destination for an I/O data.
- The source stream that provides data to the program is called the **input stream** and the destination stream that receives output from the program is called the **output stream**.
- The C++ I/O system contains a hierarchy of stream classes used for input and output operations. These classes are declared in the header file **iostream**.
- **cin** represents the input stream connected to the standard input device and **cout** represents the output stream connected to the standard output device.
- The **istream** and **ostream** classes define two member functions **get()** and **put()** to handle the single character I/O operations.
- The >> operator is overloaded in the **istream** class as an extraction operator and the << operator is overloaded in the **ostream** class as an insertion operator.
- We can read and write a line of text more efficiently using the line oriented I/O functions **getline()** and **write()** respectively.
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.
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.

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.

![Diagram of file stream classes](image)

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

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

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 use:

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

**Table 11.1 Details of file stream classes**

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

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:

```cpp
text
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.

![Diagram](image)

**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).

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

The program may contain statements like:

```cpp
text
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:

```cpp
text
Program1
.....
.....
```
ofstream outfile("salary"); // creates outfile and connects
// "salary" to it

......
......
Program2
......
...... ifstream infile("salary"); // creates infile and connects
// "salary" to it

......

Program 1

......
......

Fig. 11.5 \(\Rightarrow\) 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 on a file. Example.

......
......

outfile.close(); // Disconnect salary from outfile
ifstream infile("salary"); // 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 to the file) and `infile` (to get data from the file). Note that the use of a statement like

```cpp
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.

```cpp
// 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 keyboard and
    outf << name << "\n"; // write to file ITEM

    cout << "Enter item cost: ";
    float cost;
    cin >> cost; // get cost from keyboard 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
    inf >> cost; // read cost from file ITEM

    // Display
    cout << "name: ");
    cout << name; // Output name
    cout << "cost: ");
    cout << cost; // Output cost

    return 0;
}
```

(Contd)
cout << "\n";
cout << "Item name:" << name << "\n";
cout << "Item cost:" << cost << "\n";

inf.close(); // Disconnect ITEM from inf
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

cautions

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:

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

```cpp
WORKING WITH MULTIPLE FILES

// Creating files with open() function

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

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

    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";

    while(fin) // check end-of-file
    {
        fin.getline(line, N); // read a line
        cout << line; // display it
    }

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

(Contd)
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

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.

```c
#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
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

```cpp
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:

```cpp
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:

```cpp
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>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios::app</td>
<td>Append to end-of-file</td>
</tr>
<tr>
<td>ios::ate</td>
<td>Go to end-of-file on opening</td>
</tr>
<tr>
<td>ios::binary</td>
<td>Binary file</td>
</tr>
<tr>
<td>ios::in</td>
<td>Open file for reading only</td>
</tr>
<tr>
<td>ios::nocreate</td>
<td>Open fails if the file does not exist</td>
</tr>
<tr>
<td>ios::nolink</td>
<td>Open fails if the file already exists</td>
</tr>
<tr>
<td>ios::out</td>
<td>Open file for writing only</td>
</tr>
<tr>
<td>ios::trunc</td>
<td>Delete the contents of the file if it exists</td>
</tr>
</tbody>
</table>

**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:

```cpp
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.

![Diagram of file pointers](image)

**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

```cpp
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:

```cpp
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 now 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](image)

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

```cpp
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

<table>
<thead>
<tr>
<th>Seek call</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>fout.seekg(0, ios::beg);</td>
<td>Go to start</td>
</tr>
<tr>
<td>fout.seekg(0, ios::cur);</td>
<td>Stay at the current position</td>
</tr>
<tr>
<td>fout.seekg(0, ios::end);</td>
<td>Go to the end of file</td>
</tr>
<tr>
<td>fout.seekg(m, ios::beg);</td>
<td>Move to (m + 1)th byte in the file</td>
</tr>
<tr>
<td>fout.seekg(m, ios::cur);</td>
<td>Go forward by m byte from the current position</td>
</tr>
<tr>
<td>fout.seekg(-m, ios::cur);</td>
<td>Go backward by m bytes from the current position</td>
</tr>
<tr>
<td>fout.seekg(-m, ios::end);</td>
<td>Go backward by m bytes form the end</td>
</tr>
</tbody>
</table>

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 `<<`.

I/O OPERATIONS ON CHARACTERS

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

int main()
{
    char string[80];
    cout << "Enter a string \n";
    cin >> string;

    // (Contd)
```
for(int i=0; i<4; i++)   // clear array from memory
    height[i] = 0;

ifstream infile;
infile.open(filename);

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;
}

PROGRAM 11.5

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 reading and reading from the disk files 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 size operator. This length represents the sum total of lengths of all data members of the object.

READING AND WRITING CLASS OBJECTS

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

(Contd)
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
{
    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]));
    }

    (Contd)
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.

11.8 Updating a File: Random Acess

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.
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))
{


(Contd)
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>
<thead>
<tr>
<th>Function</th>
<th>Return value and meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>eof()</td>
<td>Returns true (non-zero value) if end-of-file is encountered while reading; Otherwise returns false(zero)</td>
</tr>
<tr>
<td>fail()</td>
<td>Returns true when an input or output operation has failed</td>
</tr>
<tr>
<td>bad()</td>
<td>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.</td>
</tr>
<tr>
<td>good()</td>
<td>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.</td>
</tr>
</tbody>
</table>

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
```
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. 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:

```c
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:

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

Program 11.8 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.
```cpp
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

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

    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 11.8 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.
- While opening the file using constructor, we need to pass the desired filename as a parameter to the constructor.
- The `open()` function can be used to open multiple files that use the same stream object. The second argument of the `open()` function called file mode, specifies the purpose for which the file is opened.
If we do not specify the second argument of the `open()` function, the default values specified in the prototype of these class member functions are used while opening the file. The default values are as follows:

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

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

To open an existing file for updating without losing its original contents, we need to open it in an append mode.

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. We can specify more than one file modes using bitwise OR operator while opening a file.

Each file has associated two file pointers, one is called input or get pointer, while the other is called output or put pointer. These pointers can be moved along the files by member functions.

Functions supported by file stream classes for performing I/O operations on files are as follows:

- `put()` and `get()` functions handle single character at a time.
- `write()` and `read()` functions write and read blocks of binary data.

The class `ios` supports many member functions for managing errors that may occur during file operations.

File names may be supplied as arguments to the `main()` function at the time of invoking the program. These arguments are known as command-line arguments.

### Key Terms

- append mode
- `argv`
- argument counter
- argument vector
- `argv`
- `bad()`
- `binary data`

- binary format
- `char` format
- `clear()`
- `command-line`
- `eof()`
- `fail()`