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 10.2 ios *format functions*

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

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 10.3 *Manipulators*

Manipulators	*Equivalent ios function*
setw()	**width()**
setprecision()	**precision()**
setfill()	**fill()**
setiosflags()	**setf()**
resetiosflags()	**unsetf()**

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()

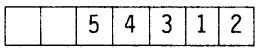
We can use the **width()** function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object to invoke it, as shown below:

```
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

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

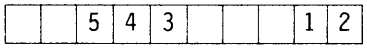
will produce the following output:



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:

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

This produces the following output:



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.

SPECIFYING FIELD SIZE WITH width()

```
#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 << "\n Grand Total = ";

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

  return 0;
}
```

PROGRAM 10.4

The output of Program 10.4 would be:

```
ITEMS          COST   TOTAL VALUE

   10            75          750
    8           100          800
   12            60          720
   15            99         1485

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:

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

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

```
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:

```
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.

PRECISION SETTING WITH precision()

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

using namespace std;

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

        cout.width(10);
        cout << "VALUE";
        cout.width(15);
        cout << "SQRT_OF_VALUE" << "\n";

        for(int n=1; n<=5; n++)
        {
                cout.width(8);
                cout << n;
                cout.width(13);
                cout << sqrt(n)  << "\n";
        }
```

(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;
}
```

PROGRAM 10.5

Here is the output of Program 10.5

```
Precision set to 3 digits
       VALUE    SQRT_OF_VALUE
         1              1
         2           1.41
         3           1.73
         4              2
         5           2.24

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*

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

This will produce the following output:

T	A	B	L	E		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,

```
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:

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

For example, the statements

```
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:

+			2	7	5	·	5	0	0

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*

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

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

FORMATTING WITH FLAGS IN setf()

```
#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)

```
        // floatfield changed
        cout.setf(ios::scientific, ios::floatfield);
        cout << "\nSQRT(100) = " << sqrt(100) << "\n";

        return 0;
}
```

PROGRAM 10.7

The output of Program 10.7 would be:

```
VALUE*********SQRT 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

SQRT(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 controls 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 10.6 *Manipulators and their meanings*

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

Some examples of manipulators are given below:

```
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:

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

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

```
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:

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout << setprecision(4);
cout << setiosflags(ios::scientific);
cout << setw(10) << 123.45678;
```

note

There is a major difference in the way the manipulators are implemented as compared to the **ios** member functions. The **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 **ios** member functions rather than the manipulators. Example:

```
cout.precision(2);              // previous state
int p = cout.precision(4);      // current state;
```

When these statements are executed, **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:

```
cout.precision(p);              // p = 2
```

Program 10.8 illustrates the formatting of the output values using both manipulators and **ios** functions.

```
#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)
```

(Contd)

```
                    << setiosflags(ios::scientific) << term
                    << setw(13) << resetiosflags(ios::scientific)
                    << sum << endl;
        }
        return 0;
}
```

PROGRAM 10.8

The output of Program 10.8 would be:

```
n    Inverse_of_n      Sum_of_terms

1    1.0000e+000          1.0000
2    5.0000e-001          1.5000
3    3.3333e-001          1.8333
4    2.5000e-001          2.0833
5    2.0000e-001          2.2833
6    1.6667e-001          2.4500
7    1.4286e-001          2.5929
8    1.2500e-001          2.7179
9    1.1111e-001          2.8290
10   1.0000e-001          2.9290
```

Designing Our Own Manipulators

We can design our own manipulators for certain special purposes. The general form for creating a manipulator without any arguments is:

```
ostream & manipulator (ostream & output)
{
   .....
   .....  (code)
   .....
   return output;
}
```

Here, the *manipulator* is the name of the manipulator under creation. The following function defines a manipulator called **unit** that displays "inches":

```
ostream & unit(ostream & output)
{
        output << " inches";
        return output;
}
```

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.

USER-DEFINED MANIPULATORS

```
#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)

```
        output.fill('*');
        output.precision(2);
        output << setiosflags(ios::fixed)
               << setw(10);
        return output;
}

int main()
{
        cout << currency << form << 7864.5;

        return 0;
}
```

PROGRAM 10.9

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.

- The **ios** class contains the member functions such as **width()**, **precision()**, **fill()**, **setf()**, **unsetf()** to format the output.
- The header file **'iomanip'** provides a set of manipulator functions to manipulate output formats. They provide the same features as that of **ios** class functions.
- We can also design our own manipulators for certain special purposes.

Key Terms

- adjustfield
- basefield
- bit-fields
- console I/O operations
- decimal base
- destination stream
- field width
- **fill()**
- filling
- fixed point notation
- flags
- floatfield
- formatted console I/O
- formatting flags
- formatting functions
- **get()**
- **getline()**
- hexadecimal base
- input stream
- internal
- **ios**
- iomanip
- iostream
- istream
- left-justified
- manipulator
- octal base
- ostream
- output stream
- padding
- **precision()**
- **put()**
- **resetiosflags()**
- right-justified
- scientific notation
- **setf()**
- **setfill()**
- **setiosflags()**
- **setprecision()**
- setting precision
- **setw()**
- showbase
- showpoint
- showpos
- skipus
- source stream
- standard input device
- standard output device
- stream classes
- streambuf
- streams
- unitbuf
- **unsetf()**
- **width()**
- **write()**

Review Questions

10.1 *What is a stream?*

10.2 *Describe briefly the features of I/O system supported by C++.*

10.3 *How do the I/O facilities in C++ differ from that in C?*

10.4 *Why are the words such as* ***cin*** *and* ***cout*** *not considered as keywords?*

10.5 *How is* ***cout*** *able to display various types of data without any special instructions?*

10.6 *Why is it necessary to include the file iostream in all our programs?*

10.7 *Discuss the various forms of* ***get()*** *function supported by the input stream. How are they used?*

10.8 *How do the following two statements differ in operation?*

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

10.9 *Both* ***cin*** *and* ***getline()*** *function can be used for reading a string. Comment.*

10.10 *Discuss the implications of size parameter in the following statement:*

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

10.11 *What does the following statement do?*

```
cout.write(s1,m).write(s2,n);
```

10.12 *What role does the* ***iomanip*** *file play?*

10.13 *What is the role of* ***file()*** *function? When do we use this function?*

10.14 *Discuss the syntax of* ***set()*** *function.*

10.15 *What is the basic difference between manipulators and* ***ios*** *member functions in implementation? Give examples.*

10.16 *State whether the following statements are TRUE or FALSE.*

(a) *A C++ stream is a file.*

(b) *C++ never truncates data.*

(c) *The main advantage of* ***width()*** *function is that we can use one width specification for more than one items.*

(d) *The* ***get(void)*** *function provides a single-character input that does not skip over the white spaces.*

(e) *The header file* ***iomanip*** *can be used in place of iostream.*

(f) *We cannot use both the C I/O functions and C++ I/O functions in the same program.*

(g) *A programmer can define a manipulator that could represent a set of format functions.*

10.17 *What will be the result of the following programs segment?*

```
for(i=0.25; i<=1.0; i=i+0.25)
{
    cout.precision(5);
    cout.width(7);
    cout << i;
    cout.width(10);
    cout <<i*i<< "\n";
}
cout << setw(10) << "TOTAL ="
     << setw(20) << setprecision(2) << 1234.567
     << endl;
```

10.18 *Discuss the syntax for creating user-defined manipulators. Design a single manipulator to provide the following output specifications for printing float values:*

(a) *10 columns width*
(b) *Right-justified*
(c) *Two digits precision*
(d) *Filling of unused places with **
(e) *Trailing zeros shown*

Debugging Exercises

10.1 To get the output Buffer1: Jack and Jerry Buffer2: Tom and Mono, what do you have to do in the following program?

```
#include <iostream.h>
void main()
{
      char buffer1[80];
      char buffer2[80];

      cout << "Enter value for buffer1 : ";
      cin >> buffer1;
      cout << "Buffer1 : " << buffer1 << endl;

      cout << "Enter value for buffer2 : ";
      cin.getline(buffer2, 80);
      cout << "Buffer2 : " << buffer2 << endl;
}
```

10.2 Will the statement cout.setf(ios::right) work or not?

```
#include <iostream.h>
void main()
{
      cout.width(5);
      cout << "99" << endl;

      cout.setf(ios::left);
      cout.width(5);
      cout << "99" << endl;

      cout.setf(ios::right);
      cout << "99" << endl;
}
```

10.3 State errors, if any, in the following statements.

(a) `cout << (void*) amount;`
(b) `cout << put("John");`
(c) `cout << width();`
(d) `int p = cout.width(10);`
(e) `cout.width(10).precision(3);`
(f) `cout.setf(ios::scientific,ios::left);`
(g) `ch = cin.get();`
(h) `cin.get().get();`
(i) `cin.get(c).get();`
(j) `cout << setw(5) << setprecision(2);`
(k) `cout << resetiosflags(ios::left |ios::showpos);`

Programming Exercises

10.1 *Write a program to read a list containing item name, item code, and cost interactively and produce a three column output as shown below.*

NAME	CODE	COST
Turbo C++	1001	250.95
C Primer	905	95.70
.....	...	.....
.....	...	.....
.....	...	.....

Note that the name and code are left-justified and the cost is right-justified with a precision of two digits. Trailing zeros are shown.

This is illustrated in Fig. 11.1.

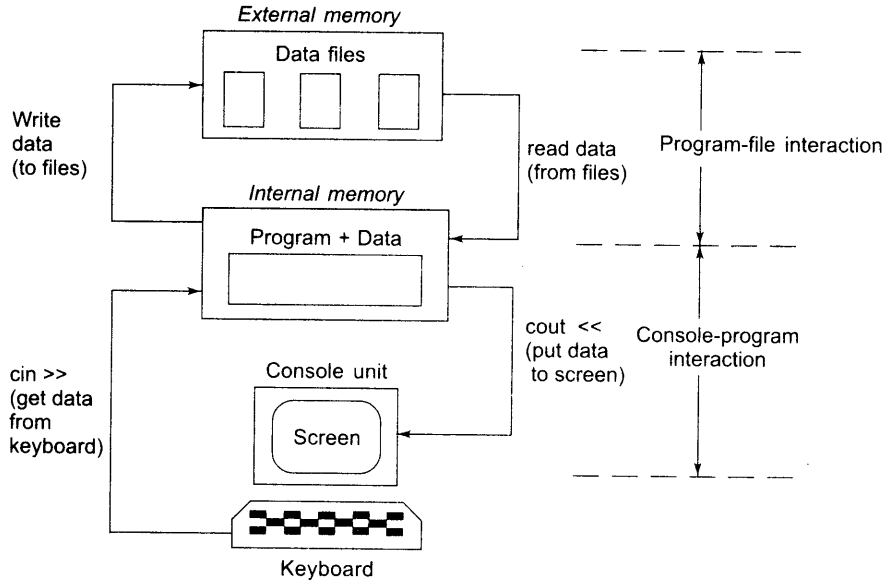


Fig. 11.1 ⇔ *Consol-program-file interaction*

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.

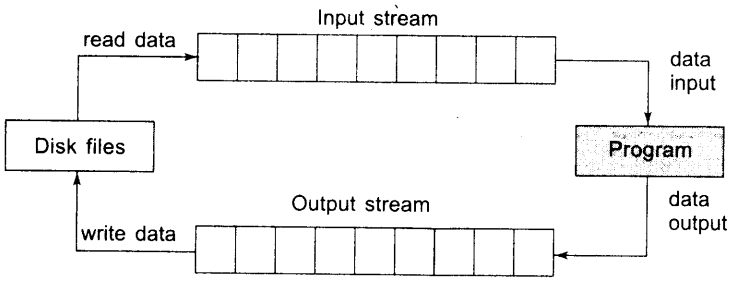


Fig. 11.2 ⇔ *File input and output streams*

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.

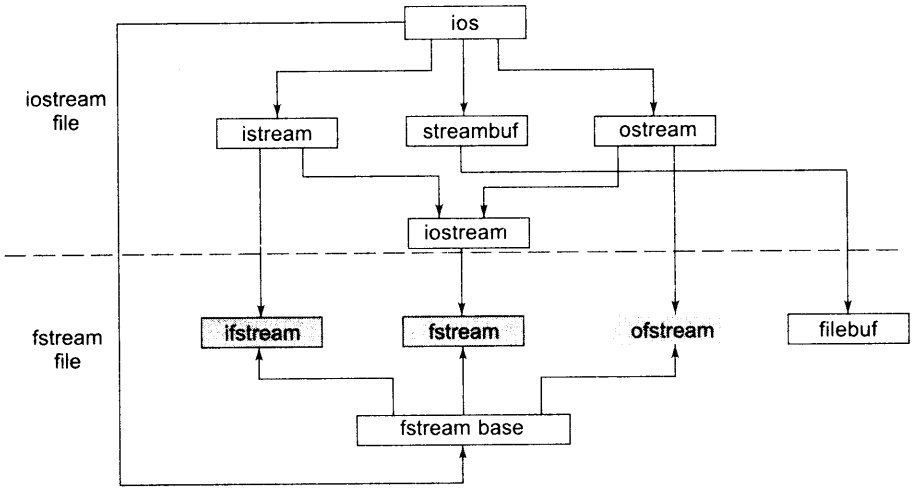


Fig. 11.3 ⇔ *Stream classes for file operations (contained in fstream file)*

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.

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

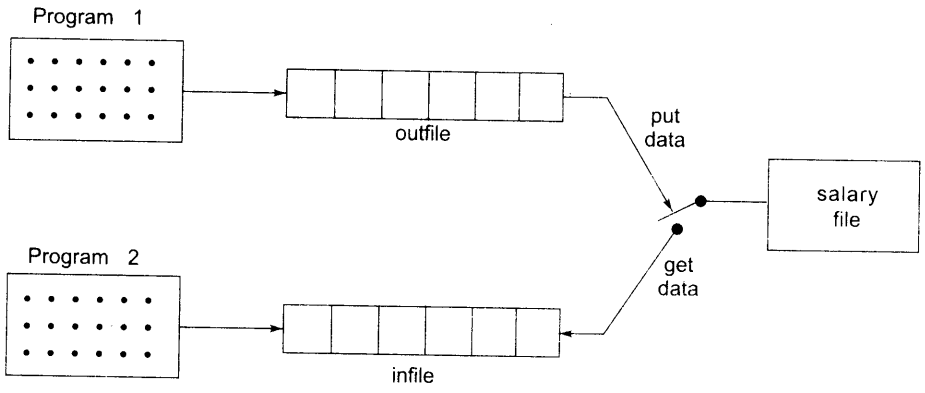


Fig. 11.5 ⇔ *Two file streams working on one file*

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 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

```
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.

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

(Contd)

```
    fin.open("capital");                    // connect "capital"

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

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

    return 0;
}
```

PROGRAM 11.2

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
```

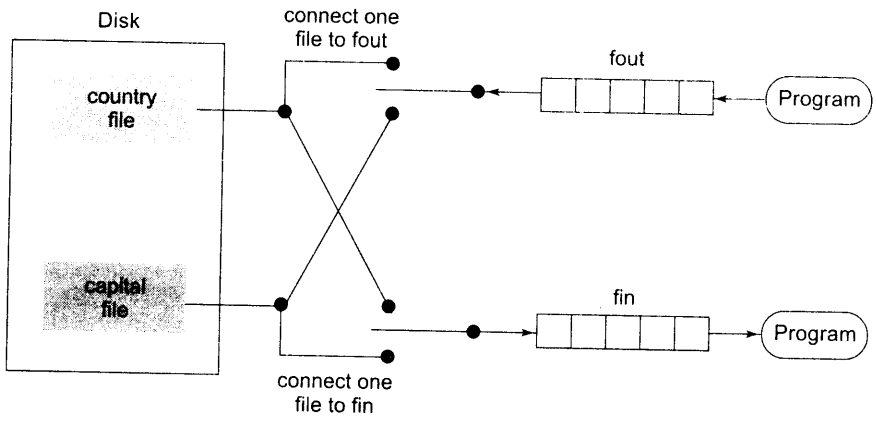


Fig. 11.6 ⇔ *Streams working on multiple files*

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.

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;
}
```

PROGRAM 11.3

The output of Program 11.3 would be:

```
Capital of United States of America
Washington
```

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.

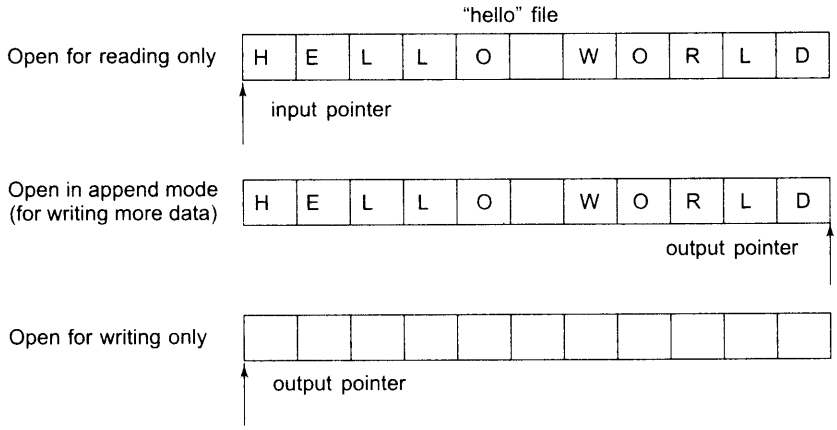


Fig. 11.7 ⇔ *Action on file pointers while opening a file*

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 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.

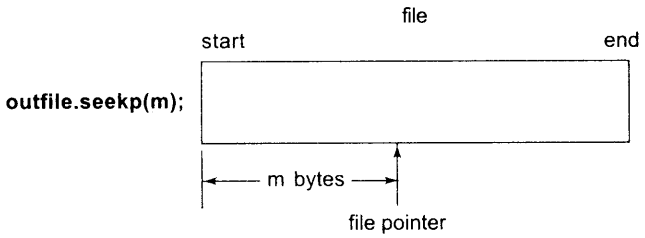


Fig. 11.8 ⇔ *Action of single argument seek function*

'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.

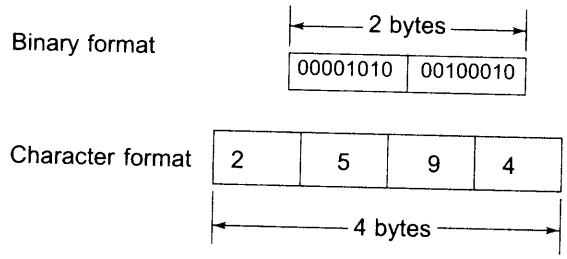


Fig. 11.9 ⇔ *Binary and character formats of an integer value*

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.

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
```

(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 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.

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)

```
        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;
}
```

PROGRAM 11.6

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.

- 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 11.7 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.

FILE UPDATING :: RANDOM ACCESS

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

class INVENTORY
{
        char name[10];
        int  code;
        float cost;
```

(Contd)

```
  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
```

(Contd)

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.