UNIT 1 STREAMS AND FILES

Structure **Page Nos.**

1.0 INTRODUCTION

In the previous blocks, we have discussed about the basic approach of object oriented programming and its important themes such as classes, inheritance, overloading, polymorphism and virtual functions. You might have seen a number of programming examples where you used **cin** and **cout** with operators >> and << for the input and output operations. We have, however, not taken up the issue of input and output to C++ programs formally. This was deliberately deferred to this point since I/O functions in C++ makes use of features like classes, derived classes and virtual functions. As we have already covered these topics, this unit builds further upon the preliminary introduction of I/O statements in C++ programs.

Through this unit, we aim to present a systematic and formal description of input and output mechanisms employed by C++ programs. C++ provides a rich set I/O functions and operations through the concept of streams and stream classes to implement I/O operations: both console and disk. The C++ I/O system contains a hierarchy of classes (known as stream classes) that are used for reading and writing by programs. Like many popular high level languages C++ supports two types of I/O: unformatted and formatted. While unformatted I/O uses functions like put(), get(), getline(), write(); formatted I/O makes use of manipulators and user-defined functions in addition to ios class functions and flags. We have also described the file stream operations and use of buffer and pointers for manipulating files.

1.1 OBJECTIVES

At the end of the unit, you should be able to:

- understand the basic mechanism of I/O in C++ through streams;
- distinguish between unformatted and formatted I/O operations in C++;
- use functions for unformatted I/O;
- design manipulators and user-defined functions for formatted I/O;
- understand stream classes for file manipulation;
- open and close files for various I/O operations;
- manage buffer and pointers for I/O from files; and
- obtain a thorough understanding and practice of using I/O functions in C++.

1.2 C++ STREAMS AND STREAM CLASSES

In C++ I/O occurs in streams, which are sequence of bytes. A stream acts as an interface between the program and the I/O device and can acts either as a source from which the input data can be obtained or as a destination to which the output data may be sent. In input operations, thy bytes flow from a device (e.g. a keyboard, a disk drive, a network connection, etc.) to main memory. In output operations bytes flow from main memory to a device (e.g., display monitor, a printer, a disk drive, a network connection, etc.). The stream that acts as a source is called input stream whereas the stream acting as destination is called output stream. The figure below illustrates the flow of data while using streams"

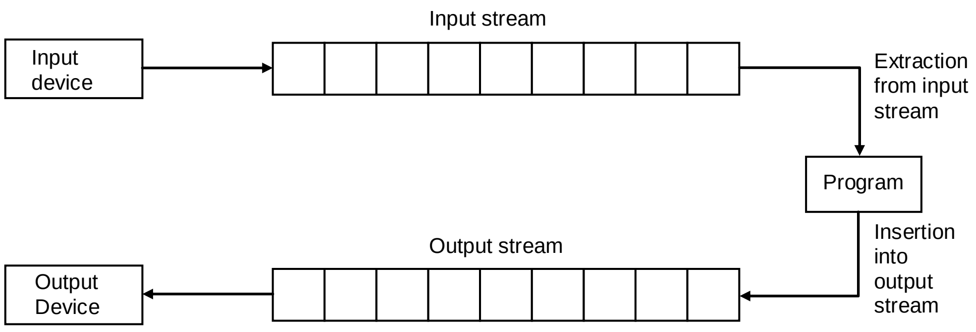

Figure 1.1 Data Streams

Though a program can take input and write output to a variety of devices (each of which may be quite different), the C++ streams provides a common interface for input and output operations irrespective of the device used. The bytes in the stream could represent characters, raw data, an image, video, speech or any other information that an application may require. It is the application (user program) that associates meaning with bytes.

In the past, the C++ used streams (often termed as classic streams) to enable input and output of characters. Since a character usually occupies one byte, it can represent only a limited set of characters (such as those in ASCII character set). Many languages however use alphabets that contain more alphabets than a single-byte character can represent. Hence C++ now includes the standard stream library that allows performing I/O operations with new character encoding systems such as Unicode. As we will shortly see, C++ now contains several pre-defined streams (e.g., cin, cout, cerr, etc.) that are automatically opened when a program begins its execution.

1.2.1 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes which are used to define various streams to manage both console and disk I/O operations. These classes are called stream classes. The figure 1.2 given below shows the hierarchy of the stream classes used for input and output operations with the console. The stream class hierarchy for disk I/O is described in section 1.5

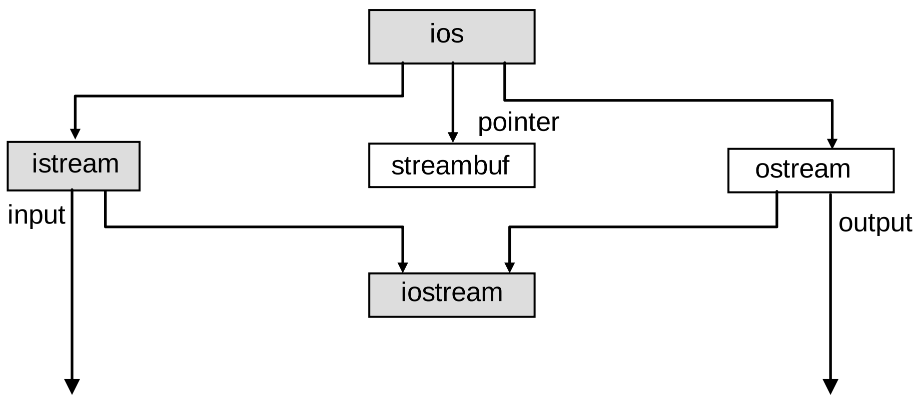

Figure 1.2 Stream Classes for Console I/O operations

The standard streams- cin, cou, cerr and clog- are also defined in <iostream> header file.

The ios class is the base class for iostream (input stream) and ostream (output stream) which are in turn base classes for iostream class (input/ output stream). The ios class is declared as the virtual base class so that only one copy of its member are inherited by the iostream. The ios class thus provides the basic support for formatted and unformatted I/O operations. The class istream provides facilities for formatted and unformatted input while the class ostream provides facilities for formatted output. The class iostream provides facilities for handling both input and output operations. All these classes are declared in the header file iostream. Most of the C++ programs include the <iostream> header file, which declares all the basic services required for all stream-I/O operations. Relevant classes for formatted I/O and disk I/O are declared in <iomanip> and <fstream> header files respectively (discussed in later part of the chapter).

1.2.2 Standard Stream Objects: cin, cout, cerr and clog

The cin, cout, cerr and clog objects correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. These are predefined objects which are by default connected to certain devices. The extraction (>>) and insertion (<<) operators define the direction of data flow. For example, the first statement below is used to read a value from keyboard and pass it to the variable 'num'; whereas the second statement the value of variable 'avg' is written to the monitor (the standard output device).

```
cin >> num;  //  reading the value of num from keyboard, data flow denotes input
cout << avg;  // displaying the value of avg on monitor, data flow denotes output
```

The number of characters read is determined by the type of variable. For example, if we type 3456X as the value for number, only '3456' is assigned to the variable 'num' and 'X' remains in the stream. Hence the operator >> reads the data character by character and assigns it to the indicated variable, unless a whitespace or a character that does not match with the destination variable type is encountered. At this point, extraction from stream is terminated and the remaining characters are consumed by subsequent cin statements. The nature of extraction (>>) and insertion (<<) operators therefore allows use of following kinds of cin and cout statements.

```
cin >> var1 >> var2 >> var3 >> ….>> varN
cout << var1 << var2 << var3 <<….<< varN
```

The variables in first statement may be variables of any type. The variables in second statement may be variables or constants of any type.

The predefined object **cerr** is an ostream instance and is connected by default to the standard error device (usually the monitor). Outputs to object **cerr** are *unbuffered*, implying that each stream insertion to **cerr** causes its output to appear immediately. (This is appropriate since it allows notifying a user promptly about errors). The predefined object **clog** is an instance of the ostream class also connected to the standard error device. However, outputs to **clog** are *buffered*. This means that each insertion to **clog** causes its ouput to be held in a buffer until the buffer is filled or it is flushed.

1.2.3 Types of I/O

The **ios** class provides the basic support for two kinds of input and output: formatted and unformatted. The class **istream** provides the facilities for formatted and unformatted input operations whereas the **ostream** class provides the facilities for formatted and unformatted output. The **istream** class declares input functions such as **get()**, **getline()**, **read()** etc., in addition to inheriting the properties of **ios** class. It also contains overloaded extraction operator >>. The **ostream** class declares output functions such as **put()** and **write()**, in addition to inheriting properties of **ios** class. It also contains overloaded insertion operator <<. The **iostream** class inherits the properties of **ios**, **istream** and **ostream** classes through multiple inheritance and hence contains all input and output functions. This is the reason why we include the iostream class only in our programs.

1.3 UNFORMATTED I/O

We have already seen use of cin and cout objects along with overloaded operators >> and << for input and output operations, in previous blocks. C++ provides many other functions for input and output operations. We will therefore briefly review their use through some examples. Functions put(), get(), getline(), putline(), read() and write() are other commonly used functions for unformatted I/O. The following sections describe, with appropriate examples, the use of these functions for unformatted I/O operations.

1.3.1 Overloaded Operators >> and <<

As stated in section 1.2.2 earlier, the overloaded **extraction (>>)** and **insertion (<<)** operators are used with **cin** and **cout** objects for stream input and output operations, respectively. The general syntax of its usage is as:

```
cin >> variable 1 >> variable 2 >> ……. >> variable N
cout << item 1 << item 2 << ……. << item N
```

where, variable 1 to N are any valid variable types in C++ and items 1 to N are any valid variable or constants in C++. C++ takes care of reading only appropriate number and type of variables (corresponding to the type of input variable) from the stream. The insertion operator puts the unformatted output into the output stream which is then displayed on the monitor. The use of these operators with **cin** and **cout** is further illustrated through following programming example:

```
# include <iostream.h>
int main()
{
        float num1, num2, num3, sum, avg;
        cout << “Enter the three numbers:”;
        cin >> num1 >> num2 >> num3;
        sum = num1 + num2 +num3;
        avg = sum / 3;
        cout << “Sum of the numbers =” << sum;
        cout << “Average of the numbers =” << avg;
        return (0);
}
```

This program first displays a prompt for entering three numbers by using cout and then reads three numbers entered from the keyboard by using cin. The program then computes sum and average of the numbers and their values are displayed on the monitor through cout statements.

1.3.2 Using member functions get (), put (), getline (), ignore (), putback () and peek ()

Unformatted input and output operations can also be carried out using various member functions of cin and cout objects provided by the istream and ostream classes. The functions get() and put() can be used to handle single character input and output operations, respectively. The general usage syntax of get is as follows:

```
get (char *)
get (void)
```

The get (char *) function assigns the character read from the keyboard to its argument. Unlike extraction operator >>, it can read blanks spaces and newline characters. [Note that >> operator simply skips the blank spaces and newline character while extracting characters from the input stream.] The get (void) simply returns the character read from the keyboard without assigning it to any variable.

The put () member function can be used to write one character to the monitor. It may take a character constant or variable as argument. The put () function can also take an integer value as input (for ex. 65), however rather than displaying the integer value it displays its ASCII equivalent character, “A” for 65. It can be put in a loop to output a line of text character by character.

```
put (‘char constant’)
put (char variable)
```

The program below illustrates use of get () and put () functions. The program reads an input text and displays it on the output screen. It also counts the number of characters and displays it on output screen. For example, if the user enters “I love Programming”, the output screen displays the entered text “I love Programming” and “Number of characters = 18” on separate lines on the output screen.

```
# include <iostream.h>
int main()
{
        int count = 0;
        char c;
        cout << “Enter some text:” ;
        cin.get(c);
        while (c!= ‘\n’)
        {
                cout.put(c);
                count++;
                cin.get(c);
        }
        cout << "\n Number of characters = " << count << "\n";
        return (0);
}
```

The functions get() and put() can handle a single character at a time. Many practical situations however require us to read and display more than a single character, for example a line of text. The getline () member function can be used for this purpose. The general syntax of getline () function is as follows:

```
cin.getline (variable, size)
```

This function reads the character input into the variable. The reading is terminated as soon as size-1 characters are read or '\n' is encountered. The delimiter character '\n' however is discarded and not saved in the variable. For example, for the code segment:

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

if the text entered from the keyboard is "IGNOU <press RETURN>", it stores the line as "IGNOU" in the variable **name**. However, if we enter "Object Oriented Programming <press RETURN>", it stores only first 19 characters in the variable **name** as "Object Oriented Pro". Similar to get (), getline () can also read blank spaces.

The **ignore ()** member function reads and discards a designated number of characters (default = 1) or terminates upon encountering delimiter EOF. The **putback ()** member function places the character just read by get () back into the stream. The **peek ()** member function returns the next character from an input stream but does not remove the character from the stream.

1.3.3 Using member functions read (), write () and gcount ()

The other member functions used to perform unformatted I/O include read (), write () and gcount (). The member function read () inputs bytes to a character array in memory; member function write () write output bytes from character array; and member function gcount () reports the number of characters read by the last input operation. The general syntax of read () and write () functions are as follows:

```
cin.read (variable, size);
cout.write (variable, size)
```

The member function read () inputs a designated number of characters (max. = size) into a character array variable. It is different from getline () in terms of usage of array variable. The member function write () outputs the characters in the character array variable on the output screen. It however does not stops at delimiter (null char) boundary and continues to display characters even beyond the line (when size > length of array). The gcount () member function is used to report the number of characters actually read by the last read () operation. The program below demonstrates the use of read () and write () member functions. If the string entered from the keyboard is "I love Programming", it stores only "I love" in the input array variable. Function gcount () returns the value as 6.

```
# include <iostream.h>
int main()
{
        char buffer [80];
        cout << "Enter a line of text \n" ;
        cin.read(buffer, 20);
        cout.write (buffer, cin.gcount());
        return (0);
}
```

☞ Check Your Progress 1

1) Fill in the blanks:

   a) Input/ output in C++ occurs as ……………………of bytes.

   b) Most C++ program that do I/O should include the ……………………… header file that contains the required for all stream I/O operations.

   c) The symbol for stream extraction operator is …………………. .

   d) The four objects that correspond to the standard devices on the system include ……………., ………………., …………………, and …………………… .

   e) Unformatted output member functions are provided in the class ………………..

2) State whether following are *True* or *False*.

   a) The cin stream normally is connected to the display screen.

   b) Input with stream extraction operator >> always skips leading blank spaces in the input stream, by default.

   c) The ostream member function put () outputs the specified number of characters.

3) For each of the following, write a single statement that performs the indicated task:

   a) Output the string "Enter your name".

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   b) Use istream member function read to input 50 characters into char array line.

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   c) Get the value of next character to input without extracting it from the stream.

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

d) Use the istream member function gcount to determine the number of characters input into character array line by the last call to read and output that number of characters using write.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

1.4 FORMATTING OUTPUTS

C++ provides a number of features that can be used display the outputs in a specified manner (formatted output). These features can be broadly categorized into following three types:

- **ios** class functions and flags
- Stream manipulators
- User-defined output functions

The **ios** class contains a large number of member functions that are used to format outputs in variety of ways. Most of the stream manipulators provide roughly the same features as that of **ios** class formatting functions, though they are at times convenient to use than their counterpart **ios** class functions. C++ allows users to design their own stream manipulators as well.

1.4.1 ios class format functions and flags

The **ios** class contains functions for defining field width, setting precision, filling and padding, displaying sign of numerical values etc. The Table 1.1 shows the lists of important **ios** class functions for formatting I/O:

Table1.1: List of important ios class functions for formatting I/O

Function	**Purpose**
width ()	To specify field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal sign
fill ()	To specify a character that fills the unused portion of an output data filed
setf()	To specify format flags such as left-justify, right-justify etc.
unsetf ()	To clear/ reset defined flags

Setting field width

The **width ()** function is used to define the width of a field required for displaying an output item. It is invoked by **cout** object as follows:

cout.width (w)

where **w** is the number of columns (field width). The output value is printed in a field of **w** characters wide at right end. This can specify field width for displaying only one output value (the one that immediately follows it). The following statements demonstrate the use of width ():

```
cout.width (5);
cout << 123 << "\n";
cout.width (5);
cout << 35;
```

The output of the above statements would be displayed in a field width of 5 as follows:

		1	2	3
			3	5

However, C++ never truncates the display value if the specified width is smaller than required. In turn it expands field width to accommodate the output value. This feature can be used to print large number of numerical values in a predetermined manner (a situation that is often encountered in accounting and other commercial applications).

Setting Precision

In C++ the floating point numbers are by default printed with six digits after the decimal point. We can however change the number of digits to be displayed after the decimal sign by using precision () function. The syntax is as follows:

cout.precision (d)

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

```
cout.precision (3);
cout << sqrt (2) << "\n";
cout << 3.14159 << "\n";
cout << 1.50009 << "\n";
```

will produce the following output:

```
1.141    (truncated)
3.142    (rounded to nearest cent)
1.5      (no trailing zeros)
```

The **precision ()** function is different from **width ()** in its effect, as the effect of precision once set continues in all subsequent statements until it is reset. The **precision ()** function can be used with **width ()** function as illustrated in following example:

```
cout.precision(2);
cout.width(5);
cout << 5.6705;
```

produces following output:

	5		6	7

The statements print the output value with a precision of 2 digits after the decimal place within a field width of 5.

Filling and Padding

The **fill ()** function is used to fill unused portion of a display field with a desired character rather than the blank spaces printed by default. The syntax is:

```
cout.fill (ch);
```

where, ch is the character to be used to fill the unused portions of a display field. For example, the following statement:

```
cout.fill (‘*’);
cout.width(10);
cout << 1234 << “\n”;
```

produces following output:

*	*	*	*	*	*	1	2	3	4

This kind of padding is very useful for institutions like banks which use it in their financial instruments (demand drafts etc) so that no one can change the figures. The **fill ()** function also stays in effect till it is reset.

Formatting Flags, Bit-fields and setf ()

The **setf ()** is another important **ios** class function that is commonly used for formatting outputs. The general syntax of **setf ()** is as follows:

cout.setf (arg1, arg2)

or

cout.setf (arg)

Here the **arg1** is one of the formatting flags defined in **ios** class and **arg2** is an **ios** constant that specifies the group to which the formatting flag belongs. These flags and bit-fields can be used for changing the alignment of display (left, right and center justify), displaying numbers in desired notation (scientific or fixed point), displaying numbers in different base systems (decimal, octal and hexadecimal). The setf () function can be used with single argument as well. In that case only flags are used without bit-fields.

The flags, bit-fields and their format actions are described in Table 1.2

Table1.2: Flags, bit-fields, formats

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	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

The **ios** flags that do not have bit-fields are described in Table 1.3:

Table 1.3: ios Flags

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

These flags are used with **setf ()** to produce desired outputs. The following example code segments demonstrate use of some of these flags:

Example Code Segment 1:

```
cout.fill ('*');
cout.setf (ios::left, ios::adjustfield);
cout.width (10);
cout << "OUTPUT 1" << "\n";
```

will produce the following output:

O	U	T	P	U	T		1	*	*

Example Code Segment 2:

```
cout.fill ('#');
cout.precision (2);
cout.setf (ios::internal, ios::adjustfield);
cout.setf (ios::scientific, ios::floatfield);
cout.width (12);
cout << -32.234 << "\n";
```

will produce the following output: (Sign is left justified and value is right-justified)

-	#	#	3	2	.	2	3	e	+	0	1

Example Code Segment 3:

```
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 << 123.4 << "\n";
```

will produce the following output: (output with sign and trailing zeros)

+			1	2	3	.	4	0	0

1.4.2 Stream Manipulators

C++ defines a number of functions called manipulators in header file **iomanip** that can be used to manipulate the output formats. Most of these manipulators are quite similar in function to the **ios** class functions and flags, though at times they are more convenient to use. The best thing is that two or more manipulators can be used in a single statement as:

cout << manip1 << manip2 << manip3 << manip4 << item;

Some of the commonly used stream manipulators and their effect is listed in Table 1.4.

Table1.4: List of Used Stream Manipulators

Manipulator	**Purpose**
setw (int w) setprecision (int d)	Set the field width to w. Set the floating point precision to d.
setfill (int c)	Set the fill character to c.
setiosflags (long f)	Set the format flag f.
resetiosflags (long f)	Clear the format flag f.
endif	Insert a new line and flush stream.

You can easily notice that in terms of functionality, **setw ()** has similar effect to ios function **width ()**, **setprecision ()** to ios function **precision ()**, **setfill ()** to ios function **fill ()**, **setiosflag ()** to ios function **setf ()**, **resetiosflags ()** to ios function **unsetf ()** and **endif** to **"\n"**. There is however a major difference between implementation of **ios** functions and stream manipulators. The **ios** member functions return the previous format state which can be used later but the manipulator does not return to previous format state.

Some example code segments illustrating use of various stream manipulators and their outputs are given below:

Example Code Segment 1:

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

will produce the following output: (12345 is displayed right-justified in a field width of 10)

					1	2	3	4	5

Example Code Segment 2:

```
cout << setw (5) << setprecision (2) << 1.2345;
cout <<  setw (10) << setprecision (4) << 3.1415435;
```

will produce the following output: (Sign is left justified and value is right-justified)

	1	.	2	3

				3	.	1	4	1	5

1.4.3 User Defined Stream Manipulators

C++ allows users to design their own customized stream manipulators. These user-defined manipulators may be non-parameterized or parameterized ones. The general syntax for creating a user-defined stream manipulator without any argument is as follows:

```
Ostream & manipulator (ostream & output)
{
          …………
          ………… Statements for formatting
          …………
          return output;
}
```

Here, the manipulator is the name of user-defined manipulator to be created. For example, we can create the user defined manipulator unit that displays “INR” after each numerical value displayed.

```
ostream & unit (ostream & output)
  {
   output << “INR”;
   return output;
  }
```

A more complex user-defined manipulator show can be defined as follows:

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

The following program presents a full blown example of designing user-defined stream manipulators. This program creates two user-defined manipulators currency and form, which are used to display output values in a specific manner.

```
# include <iostream.h>
#include <iomanip.h>
ostream & currency(ostream & output)
{
        output << “INR”;
        return (output);
}
ostream & form(ostream & output)
{
        output.setf (ios::showpos);
        output.setf (ios::showpoint);
        output.fill (‘*’);
        output.precision (2);
        output << setiosflags (ios::fixed) << setw (10);
        return (output);
}
```

```
int main()
{
        cout << currency << form << 5465.4;
        return (0);
}
```

The output of the program would be INR **+5465.40.

Another example code which makes use of stream manipulators and other formatting techniques is given below. This program creates two user-defined manipulators area and volume, which are used to display output values in a specific manner. The value of area is displayed with SQ.MTS unit and the volume is displayed with CUBIC MTS unit.

```
# include <iostream.h>
#include <iomanip.h>
ostream & area(ostream & output)
{
        output << “SQ.MTS”;
        return (output);
}
ostream & volume(ostream & output)
{
        output.precision (4);
        output << setiosflags (ios::fixed) << setw (15);
        output << “CUBIC MTS”
        return (output);
}
int main()
{
        cout << area << 3000;
        cout<< volume << 9000;
        return (0);
}
```

☞ Check Your Progress 2

1) Fill in the blanks:
   - a) Member function ……………. can be used to set and reset format state.
   - b) The stream manipulators that format justification are ………………., ………………., and ………………. .
   - c) The ……………. stream manipulator causes positive number to be displayed with a plus sign.
   - d) The functionally equivalent ios function for stream manipulator setw () is ……………….

2) State whether True or False:
   - a) The stream manipulators dec, oct and hex affect only the next integer output operations.
   - b) The stream member function flags with a long argument sets the flags state variable to its argument and returns its previous value.
   - c) By default, memory addresses are displayed in hexadecimal format.

3) For each of the following, show the output:

a) cout << setw (10) << setfill (‘$’) << 10000;

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

b) cout << showbase << oct << 99 << endl << hex << 99;

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

c) cout << 10000 << endl << showpos << 10000;

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4) For each of the following, write a single C++ statement that performs the desired task.

a) Use a stream manipulator such that, when integer values are output, the integer base for octal and hexadecimal values is displayed.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

b) Print the current precision setting, using a member function of object cout.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

c) Print 1234 right justified in a 10- digit field.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

d) Print 1.92, 1.925 and 1.9258 separated by tabs and with 3 digits of precision, using a stream manipulator.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

5) Create a user-defined manipulator *showcurr* that prints '#' sign before every numerical value displayed.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

1.5 FILE STREAM OPERATIONS

Many real world applications require reading and writing large number of data items. Usually large volumes of data are stored as files on disk. Like many high level programming languages, C++ also provides mechanism to read and write data items from files. A C++ program can thus take input from a disk file and also write data to it. C++ file handling mechanism is quite similar to console input-output operations. It uses streams (called file streams) as an interface between programs and files. The Figure 1.3 illustrates the use of file streams for input and output:

The stream that supplies data to the program is called input stream and the one receiving data from program for writing to file is called output stream.

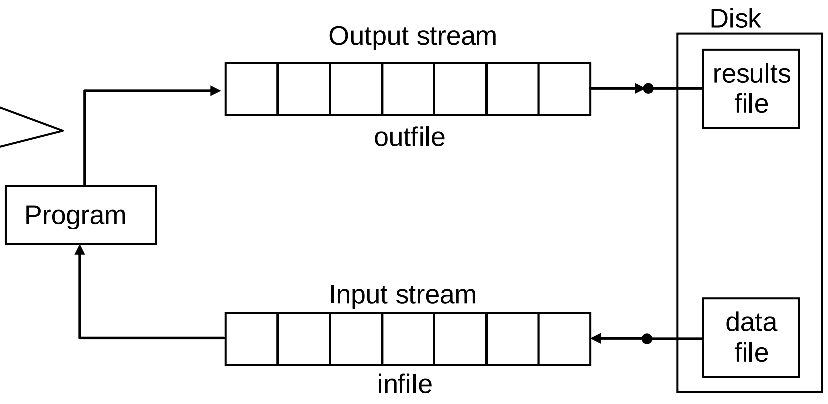

Figure 1.3 File Input and Output Steams

The Figure 1.3 shows a C++ program that reads data from one file and writes the output to another file (named results). The input stream is connected to data file used for input and output stream is connected to results file used for output. Programs can do reading and writing on the same file as well.

The I/O system of C++ contains a set of classes that provide methods for reading and writing from files. These classes are ifstream, ofstream and fstream. All these classes are derived from fstream base class and also inherits features from iostream class. The stream classes and their inheritance is shown in the Figure 1.4:

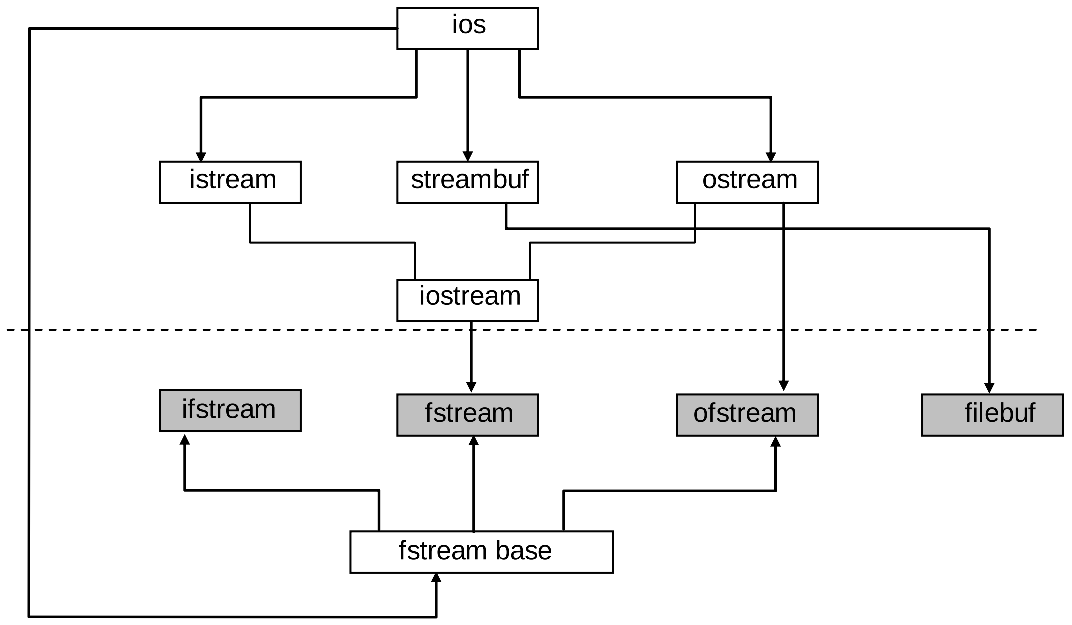

Figure 1.4 File Steam Classes

All these classes are declared in **fstream** and that is why this header file is included in all programs doing file processing. The **fstream** base class provides operations common to the file streams and contains **open()** and **close()** functions. It also serves as base class for the other three file stream classes. The **ifstream** class provides functions for input operations. This includes functions like **get()**, **getline()**, **read()**, **seekg()** etc. The **ofstream** class provides functions for output operations and include functions like **put()**, **write()** etc. The **fstream** class provides support for simultaneous input and output operations. It also inherits all functions from **istream** and **ostream** classes through **iostream**. The **filebuf** is used for setting the file buffers for read and write operations. This is required since volume of data is read and written to files.

Opening and Closing Files

Every disk file has a name (usually a string of valid characters). In order to read data from a file or to write data into it, we first need to open the file. This opened file has to be then connected to input-output stream so that the corresponding program becomes able read and write from it. A file stream can be defined using any of the three classes ifstream, ofstream or fstream, depending upon the purpose of the file.

A file can be opened either by using the **constructor** function of the class or by invoking member function **open ()** of the class. In order to open file using constructor function, we may use statements of the form:

```
ifstream infile (“data”); and
ofstream outfile (“results”);
```

Here, first statement opens a file called data and attaches it to the stream infile. This file can then be used for input operations. The second statement opens a file called results and attaches it to stream outfile. This file can only be used for output operations. Once the infile and outfile streams are created and connected to corresponding files, statements of the form:

```
outfile << "Sum";
outfile << "avg";
infile >> num;
infile >> name;
```

can be used to write data items to the output file results and to read data from input file data. After completing the input-output operations, the file may be closed by closing the corresponding stream. For example:

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

Even if we fail to explicitly close a file, it gets closed automatically when the program terminates (and hence the corresponding stream expires). Nevertheless it is a good practice to close the open files once they are no longer required.

A file can also be opened using **open ()**. The general syntax of open () is as follows:

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

For example, to open a file data for input, we may use the statements:

```
ifstream infile;
infile.open("data");
```

Now, the stream infile can be used for reading data form the file "data" in a similar manner as that of console I/O. After the file's use is complete, it should be closed by closing the corresponding (by invoking close() function).

C++ allows the files to be opened in different modes. Hence, the open() function can specify the mode of opening as well. The statement invoking open can be written as:

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

The mode argument specifies the purpose for which the file is opened. The file mode parameter is defined in ios class and can take any of the following values:

Mode Parameter	**Effect**
ios::app	Append to end-of-file.
ios::ate	Go to end-of-file on opening.
ios::binary	Open a binary file.
os::in	Open file for input only.
ios::nocreate	Open fails if file does not exist.
ios::noreplace	Open already existing file.
ios::out	Open file for output only.
ios::trunk	Delete the contents of the file if it exists.

The function open() contains default values for these modes and hence even if the mode is not specified, the file is opened with default mode.

One must also be careful in reading from an input file so as to ensure that no read attempt is made once the file end is reached. We will see the use of EOF() member function of ios class for this purpose in next section.

An example program demonstrating how we can read from a disk file and write the results to the disk file is given below:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
        const int SIZE = 80;
        char line [SIZE];
        ifstream infile;
        ofstream outfile;
        infile.open(“namelist”);
        outfile.open(“results”);
        while(infile.eof()!=0)
        {
                infile.getline(line, SIZE);
                outfile << line;
        }
        return (0);
}
```

This program opens a file “namelist” for reading and “results” for writing. Then it continues to read the entire contents of file “namelist” and copies it to the file “results”. The getline() stream member function is used for reading and the output content is simply directed to the stream “outfile” for writing it to the file “results”. The reading (and subsequent copying) continues till the end of file “namelist” is reached. If one needs to display the contents while copying, a statement cout << line; may be added. This program can be made meaningful, if we copy the sorted namelist to the “results” file. This can be done by first sorting the contents of “namelist” file (temporarily storing it in an array of strings) and then writing the sorted values.

1.6 FILE POINTERS AND OPERATIONS

However, if we open a file in append mode, the output pointer is set to the end of file.

We can further control the reading and writing operations from file by manipulating the file pointers. Every file in C++ is marked by two pointers, one for input (called get pointer) and one used for output (called put pointer). The get pointer can be set to a specified location in order to reach from that desired location in the file. These pointers are automatically incremented every time after every read and write operation. You may be wondering how we have been able to read and write in our earlier programs without setting these pointers. Actually every time we open a file for input, the input pointer is automatically set to the beginning of the file.

We can manipulate these file pointers by invoking member functions of the file stream class. The commonly used functions for this purpose include **seekg()**, **seekp()**, **tellg()**, **tellp()** etc. The **seekg()** function moves the input (get) pointer to a specified location. The **seekp()** function moves to output (put) pointer to a specified location. The **tellg()** and **tellp()** functions tell the current position of get and put pointers, respectively. The general syntax for using **seekg()** and **seekp()** is given below:

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

Where, **offset** represents the number of bytes, the file pointer is to be moved from the location specified by the parameter refposition. The **refposition** may take one of the three positions defined in the **ios** class:

ios::beg – start of the file
ios::cur – current position of the pointer
ios::end – end of file.

The seekg() function moves the get pointer whereas the seekp() function moves the put pointer as per the parameters specified in the statement.

Functions to read from and write to files

Reading and writing operations in files are made simpler by C++ by providing some member functions in the file stream class. The functions **put()** and **get()** are used for handling a single character at a time. Functions **read()** and **write()** are designed to handle blocks of binary data. We also have functions like **getline().**

Two programming examples demonstrating use of **put()** – **get()** and **read() - write()** are given below:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main()
{
        char name [80];
        cin >> name;
        int len = strlen(name);
        fstream file;
        file.open(“text”, ios::in I ios:out);
        for (int i =0; I < len; i++)
                file.put(name[i]);
        file.seekg(0);
        char c;
        while (file)
        {
                file.get(ch);
                cout << ch;
        }
        return (0);
}
```

The program above reads a string and puts it character by character in a file called “text”. This file is opened for both reading and writing. Then the program goes to read the contents written in the file one character at a time by using get() function. Before doing that, the file pointer is set to the beginning of the file.

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename = “BINARY”;
int main()
{
        int num[5] = {12, 23, 34, 45, 56};
        ofstream outfile;
        outfile.open(filename);
        outfile.write((char *) & num, sizeof(num));
        outfile.close();
        for (int i=0; i<5; i++)
```

```
                num[i]= 0;
        ifstream infile;
        infile.open(filename);
        infile.read((char *) & num, sizeof(num));
        for (i=0; i<5; i++)
        {
                cout.setf(ios::showpos);
                cout << setw(5) << name[i];
        }
        infile.close();
        return (0);
}
```

The program above reads an integer array and then writes its contents to a binary file using write(). The file is then opened again this time for reading and the contents written in the file are read and displayed on the monitor. The read() function is used for reading.

☞ Check Your Progress 3

1) Fill in the blanks:

   a) Header file ……………… contains the declarations required for file processing.
   b) The ios mode ……………….. only opens a file that already exists, otherwise it fails.
   c) The function……………………. can be used to position the input file pointer at a desired location.

2) State whether the following statements are True or False.

   a) The file opening mode ios::ate positions the file pointer at end-of-file on opening.
   b) The function tellp() can be used to detect end of a file.
   c) A C++ program can do reading and writing on the same file.

3) For each of the following, write a single statement that performs the desired task:

   a) To open a file "text" in append mode and connect a stream infile to it.

   ………………………………………………………………………………………
   ………………………………………………………………………………………
   ………………………………………………………………………………………

   b) To read a single character in a file connected to stream file.

   ………………………………………………………………………………………
   ………………………………………………………………………………………
   ………………………………………………………………………………………

c) To read an entire line of text of *size* characters from a file connected to f1 file stream into a variable *line*.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

d) To detect the end of a file connected to stream file.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

1.7 SUMMARY

This unit has introduced you to the concept of streams and how they are used for console and disk I/O operations by a C++ program. The stream concept provides in C++ the flexibility of using the same mechanism for input-output operations from different devices. A stream is a sequence of bytes and it serves as both source and destination for an I/O data. C++ provides a hierarchy of stream classes that support different functions for managing I/O devices from console and disk. There are certain streams which are by default connected to standard devices. The console I/O operations may be unformatted or formatted. We use separate functions for both purposes. Output can be produced in a desired manner (formatted) by using one of the three mechanisms: ios class functions and flags, stream manipulators and user-defined stream manipulator functions. C++ also provides a large number of functions that can be used to read and write data from disk files. Files need to be opened and closed for I/O operations. Files may be opened in different modes depending upon the kind of operation to be performed. C++ provide different functions for manipulating file pointers and hence deciding where to read and write the data.

1.8 ANSWERS TO CHECK YOR PROGRESS

Check Your Progress 1

1. a) streams b) <iostream> c) >> d) cin, cout, cerr, clog
   e) ostream
2. a) False, It is connected to standard input, which is keyboard.
   b) True
   (b) False, It outputs its single character argument.
3. a) cout << "Enter your name";
   b) cin.read (line, 50);
   c) cin.peek ()
   d) cout.write (line, cin.gcount());

Check Your Progress 2

1. a) flags b) left, right and internal
   c) showpos d) width ()

2. a) False, They have effect till it is reset or the program terminates.
   b) False, The stream member function flags with a fmtflags argument sets the flags state variable to its argument and returns the prior state setting.

c) True

3. a) $$$$$10000
   b) o143
      oX63
   c) 10000
      +10000

4. a) cout << showbase;
   b) cout << cout.precision ();
   c) cout << setw (10) <<1234;
   d) cout << setprecision (3) << 1.92 << '\t' << 1.925 << '\t' <<1.9258

5.

```
ostream & showcurr (ostream & output)
{
output << "#";
return output;
}
```

Check Your Progress 3

1. a) <fstream>   b) ios::noreplace   c) seekg()

2. a) True
   b) False, It is actually eof() function.
   c) True
3. a) ifstream infile; infile.open("text", "ios::app");
   b) file.get(ch);
   c) f1.getline(line, size);
   d) file.eof()

1.9 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, *C++: How to Program, PHI*, 7th edition, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.