Manipulator	Action Performed	Equivalent to
`setw(int width)`	Sets the field width	`width`
`setprecision(int prec)`	Sets the floating-point precision	`precision`
`setfill(int fchar)`	Sets the fill character	`fill`
`setbase(int base)`	Sets the conversion base 0: Base 10 is used for output 8: Use octal for input and output 10: Use decimal for input and output 16: Use hexadecimal for input and output	
`setiosflags(long flags)`	Sets the format flag	`setf`
`resetiosflags(long flags)`	Resets the format flag	`unsetf`

Table 17.5: C++'s predefined parameterized manipulators

Buffering

When a stream is buffered, each insertion or extraction does not have a corresponding I/O operation to physically write to or read data from a device. Instead, insertions and extractions are stored in a buffer from which data is written or read in chunks.

In C++, it is possible to force data buffered in an output stream to be written. It is called flushing and it ensures that everything stored in an output buffer has been displayed. In general, flushing is done when interactive input is requested by the user, so that the program can be sure that information displayed on the screen is completely up-to-date. The *cout's* buffer can be flushed using the statement,

```
cout.flush();
```

A program can `tie` an input stream to an output device. In this case, the output stream is flushed when any characters are fetched from the input stream. For instance, `cin` is automatically tied to `cout` to be sure that everything has been physically displayed before any input occurs. The user defined streams can be tied using the `tie` function as follows:

```
istream input;
ostream output;
....
input.tie( output );
```

The last statement forces the C++ I/O system, to flush the object stream, output every time the fetch operation is initiated using the object, `input`.

The parameterized manipulators are described below:

`setw(int width)`: Sets the width of the output field specified by the integer parameter width. The output field width is reset to 0 every time an output is performed using the << operator. When the output field width is 0, normal output is done (without filling or aligning). Hence, use the `setw` manipulator to specify the field width before every output for which a particular field width is desired.

`setprecision(int prec)`: Sets the precision used for floating point output. The number of digits to be shown after the decimal point is given by the integer `prec`.

setfill(int fchar): Sets the *fill character* to that specified in fchar. The fill character is used to fill (or pad) the empty space in the output field when the width of the output variable is less than the width of the output field. The default fill character is the space character.

setbase(int base): Sets the conversion base according to the integer base, which can assume any one of the following four values:

0: Base 10 is used for output;
8: Use octal for input and output.
10: Use decimal for input and output.
16: Use hexadecimal for input and output.

The base to be used for input is specified as a part of the input itself - inputs beginning with 0 are treated as octal, those beginning with 0x are treated as hexadecimal. Otherwise, the base is assumed as decimal.

setiosflags(long flags): The parameter flags can be any of the flags listed in ios stream class. More than one flag can be set with the same manipulator by ORing the flags.

The statement

```
cout << setw( 8 ) << 1234;
```

prints the value 1234 right-justified in the field width of 8 characters. The output can be left justified using the statement,

```
cout << setw( 8 ) << setiosflags( ios::left ) << 1234;
```

The key difference between manipulators and the ios class interface functions is in their implementation. The ios member functions are used to read the previous format-state, which can be used to know the current state or save for future usage, whereas, the manipulators do not return the previous format state. The program foutput.cpp illustrates the use of some of the manipulators with output streams.

```
// foutput.cpp: various formatting flags with the << operator
#include <iostream.h>
#include <iomanip.h>
void main()
{
   int x = 100;
   cout << hex << x << ' ' << dec << x << endl;
   float f = 122.3434;
   cout << f << endl;
   cout << setprecision( 3 );
   cout << f << endl;
   cout << setw( 6 ) << setfill( '0' );
   cout << setiosflags(ios::internal | ios::showbase);
   cout << hex << x << endl;
   cout << setiosflags( ios::scientific ) << f << endl;
}
```

Run

```
64 100
122.343399
122.343
0x0064
1.223e+02
```

In `main()`, the statement

```
cout << hex << x << endl;
```

outputs 0x0064, since the field width 6 and the fill character '0' is filled between the base indicator '0x' (due to ios::showbase) and the number 64 (padding like this occurs due to ios::internal being set).

The program `payroll.cpp` uses the manipulators for displaying numeric quantities for accounting purposes so that the decimal points are aligned in a single column.

```
// payroll.cpp: payroll like output example
#include <iostream.h>
#include <iomanip.h>
void main()
{
  float f1=123.45, f2=34.65, f3=56;
  cout << setiosflags(ios::showpoint|ios::fixed)
       << setiosflags(ios::right);
  cout << setw(6) << f1 << endl;
  cout << setw(6) << f2 << endl;
  cout << setw(6) << f3 << endl;
}
```

Run

```
123.45
 34.65
 56.00
```

Setting the flag `ios::showpoint` will display the point even though a floating point number has no significant digits to the right of the decimal point (the variable f3). Setting `ios::fixed` ensures output in fixed point rather than in exponential notation. The decimal points happen to be aligned due to two manipulators: `setprecision(2)`— show two digits after the decimal point and `setiosflags(ios::right)`— display output in right-justified manner.

```
// oct.cpp: Usage of number-base manipulators with cin
#include <iostream.h>
#include <iomanip.h>
void main()
{
   int i;
   // The statement below always interprets the input as octal digits
   cout << "Enter octal number: ";
   cin >> oct >> i;
   cout << "Its decimal equivalent is ";
   cout << i << endl;
   //The base used by cin in the statement is decided at the time of input
   cout << "Enter decimal number: ";
   cin >> setbase( 0 ) >> i;
   cout << "Its output: ";
   cout << i;
```

Run1

```
Enter octal number: 111
Its decimal equivalent is 73
Enter decimal number: 0111
Its output: 73
```

Run2

```
Enter octal number: 111
Its decimal equivalent is 73
Enter decimal number: 0x111
Its output: 273
```

In the `cin` statement

```
cin >> oct >> i;
```

data input is always interpreted as an octal number. So, if the input is 111, the output using the `cout` statement here is 73. Whereas, in the statement

```
cin >> setbase( 0 ) >> i;
```

if the input to the `cin` statement here is 111, then it is assumed to be a decimal number. If it is 0111, it is assumed as an octal number. Finally, an input such as 0x111 is assumed hexadecimal. So the output of the last `cout` statement will be 111 in the first case, 73 in the second, and 273 in the third.

The program `mattab.cpp` illustrates the use of manipulators and `ios` functions in formatting the output.

```
// mattab.cpp: prints mathematical table having sqr, sqrt, and log columns
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
// macro for computing square of a number
#define sqr( x ) ((x)*(x))
void main()
{
   int num;
   cout << "Enter Any Integer Number: ";
   cin >> num;
   cout << "---------------------------------------------" << endl;
   cout << setw( 5 ) << "NUM" << setw( 10 ) << "SQR";
   cout << setw( 15 ) << "SQRT" << setw( 15 ) << "LOG" << endl;
   cout << "---------------------------------------------" << endl;
   cout.setf( ios::showpoint );  // display trailing zeros
   for( int i = 1; i <= num; i++ )
   {
      cout << setw( 5 ) << i
           << setw( 10 ) << sqr( i )
           << setw( 15 ) << setprecision( 3 ) << sqrt( (double) i )
           << setw( 15 ) << setprecision( 4 ) <<setiosflags(ios::scientific)
           << log( (double) i ) << endl << resetiosflags( ios::scientific );
   }
   cout << "---------------------------------------------" << endl;
```

Run

```
Enter Any Integer Number: 10
----------------------------------------------
 NUM        SQR           SQRT          LOG
----------------------------------------------
   1          1          1.000     0.0000e+00
   2          4          1.414     6.9315e-01
   3          9          1.732     1.0986e+00
   4         16          2.000     1.3863e+00
   5         25          2.236     1.6094e+00
   6         36          2.449     1.7918e+00
   7         49          2.646     1.9459e+00
   8         64          2.828     2.0794e+00
   9         81          3.000     2.1972e+00
  10        100          3.162     2.3026e+00
----------------------------------------------
```

17.7 Custom/User-Defined Manipulators

An important feature of C++ streams is that they also work well with the user-defined manipulators as they do with *built-in manipulators*. Hence, the users can design their own (customized) manipulators to control the appearance of the output depending upon their taste and need. The syntax for creating a custom manipulator is shown in Figure 17.4. In the syntax, `manipulator` is the name of the user-defined manipulator.

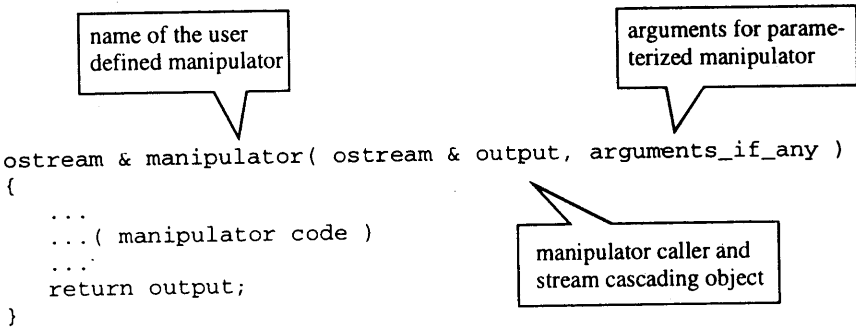

Figure 17.4: Syntax of creating a custom manipulator

The program `space3.cpp` creates and uses the user-defined manipulator `sp` that inserts space into the output stream and flushes it. It eliminates the usage of messy statements such as,

```
cout << x << ' ' << y << ' ' << z << ' ' << w << endl;
```

to output a series of variables separated by spaces. The statement can be written as,

```
cout << x << sp << y << sp << z << sp << w << endl;
```

which appears more elegant and simple to use and understand.

```
// space3.cpp: custom built manipulator
#include <iostream.h>
// The user-defined manipulator
ostream & sp( ostream& os )
{
   os << ' ' << flush;  // or cout << ' ' << flush
   return os;
}
void main()
{
   int x=1, y=2, z=3, w=4;
   cout << x << sp << y << sp << z << sp << w << endl;
}
```

Run

```
1 2 3 4
```

In the above program, the function

    ostream & sp(ostream& os)

defines a manipulator called sp that prints a single space and flushes the same to console for immediate display without buffering.

Another interesting use of manipulators is demonstrated in the program `currency.cpp`. It defines manipulators for prefixing the currency symbol to an item cost depending on the currency used by the country which has manufactured the item.

```
// currency.cpp: custom built manipulator for currency unit representation
#include <iostream.h>
// currency in Indian rupees
ostream & rupee( ostream& os )
{
   os << "Rs. " << flush;
   return os;
}
// currency unit in US dollar
ostream & dollar( ostream& os )
{
   cout << "US$ " << flush;
   return os;
}
void main()
{
   char item1[25], item2[25];
   unsigned int cost1, cost2;
   cout << "Item Sales in India..." << endl;
   cout << "Enter Item Name: ";
   cin.getline( item1, 25 );
   cout << "Cost of Item: ";
   cin >> cost1;
   cout << "Item Sales in US..." << endl;
```

```
    cout << "Enter Item Name: ";
    cin.ignore();
    cin.getline( item2, 25 );
    cout << "Cost of Item: ";
    cin >> cost2;
    cout << "Item Cost Statistics..." << endl;
    cout << "Item Name: " << item1 << endl;
    cout << "Cost: " << rupee << cost1 << endl;
    cout << "Item Name: " << item2 << endl;
    cout << "Cost: " << dollar << cost2 << endl;
}
```

Run

```
Item Sales in India...
Enter Item Name: PARAM Supercomputer
Cost of Item: 55000
Item Sales in US...
Enter Item Name: CRAY Supercomputer
Cost of Item: 40500
Item Cost Statistics...
Item Name: PARAM Supercomputer
Cost: Rs. 55000
Item Name: CRAY Supercomputer
Cost: US$ 40500
```

Standard Manipulators Implementation

The previous example was easy, since the manipulator did not accept any parameters in the output statement. The function that overloads the << operator to accept manipulators merely needs to call the manipulator with the output stream object (`cout` in this case). Manipulators accepting parameters initiates many actions. Consider the manipulator declared in `iomanip.h` header file, `setw(int)`, to illustrate the implementation of manipulators. The declaration of this manipulator is:

```
ostream & setw( ostream&, int );
```

But in the output statement, `setw` is called with only one integer argument:

```
cout << setw(6) << i;
```

Another function (also called `setw`) is needed that accepts only one argument of type integer. It does not know which output object needs to have its field-width set. Assuming the output object as `cout` will unduly restrict its use (For instance, it would not be possible to use it directly with files). To resolve this impasse, the following solution is used. A class called `omanip_int` is declared. It has two private members; a pointer to function (the actual manipulator) and an integer that specifies the width. It has a constructor that sets these members, and a friend function that overloads the << operator and calls the actual manipulator.

```
class omanip_int
{
  private:
    ostream& (*f)( ostream&, int ); // Pointer to the actual manipulator
    int w;              // Width to be set
```

```
  public:
   //Constructor
    omanip_int( ostream& (*tf)(ostream&,int), int tw )
    { f = tf;
      w = tw;
    }
    //  overloading stream output operator
    friend ostream& operator << (ostream& os, omanip_int o)
    {
      return o.f( os, o.w );      //Call the actual manipulator.
    }
};
```

Two more functions are now required; one that actually manipulates the stream, and another that is invoked from the output statement. They are declared as follows:

```
//Actual manipulator
ostream& setw( ostream& os, int w )
{
  os.width( w );
  return os;
}
// This is called first from the output statement.
// It accepts an integer and returns an instance of class omanip_int
omanip_int setw( int w )
{
  return omanip_int( setw, w ); // returns nameless object
}
```

Now, the statement

```
     cout << setw( 6 ) << i;
```

will first call the second `setw` manipulator that *remembers* the width passed in an instance of the class `omanip_int`. The actual function to be called is also recorded here. This instance is returned. The first << above now has the return value of `setw(6)` - an instance of `omanip_int` on the right, and `cout` on the left. The overloaded function (defined in the class omanip_int) is invoked, which in turn calls the actual manipulator. The same concept can be utilized while implementing the user-defined manipulators.

Parameterized Custom Manipulators

Most manipulators do not accept parameters and are simple to use. Sometimes it is necessary to pass data to the manipulators, however, as with the built-in manipulator `setw(int)`. The program presented in `pmani.cpp`, implements a manipulator that accepts three arguments - width, precision, and fill character. The manipulator is useful as a shorthand notation for setting the above parameters to output floating point variables with different width, precision, and fill characters.

```
// pmani.cpp: Parameterized Manipulator
#include <iostream.h>
#include <iomanip.h>
// output manipulator taking arguments of type int, int, char
```

```
class my_manipulator
{
   private:
      int width, precision;
      char fill;
   public:
      //Constructor
    my_manipulator(int tw, int tp, char tf):width(tw),precision(tp),fill(tf)
     {}
    //Overloaded << operator
    friend ostream & operator << ( ostream& os, my_manipulator object );
};
//Actual manipulator called by overloaded operator << friend function
ostream & operator << ( ostream& os, my_manipulator object )
{
   os << setw( object.width ) << setprecision( object.precision ) \
   << setfill( object.fill );
   os << setiosflags(ios::showpoint|ios::right);
   return os;
}
//Function called first from the output statement
my_manipulator set_float( int w, int p, char f )
{
  return my_manipulator( w, p, f ); // nameless object is returned
}
void main()
{
   float f1=123.2734, f2=23.271, f3=16.1673;
   // set_float accepts three parameters-width, precision and fill character
   cout << set_float( 10, 3, '*' ) << f1 << endl;
   cout << set_float(  9, 2, '^' ) << f2 << endl;
   cout << set_float(  8, 3, '#' ) << f3 << endl;
}
```

Run

```
***123.273
^^^^23.27
##16.167
```

In `main()`, the statement

```
cout << set_float( 10, 3, '*' ) << f1 << endl;
```

has the call to the normal function as,

```
set_float( 10, 3, '*' )
```

which in turn creates the nameless object of the class `my_manipulator` (and initializes its members) and returns the same. Thus, the above output statement effectively becomes,

```
cout << my_manipulator( set_float, 10, 3, '*') << f1 << endl;
```

The class `my_manipulator` is a friend of the overloaded operator function and hence, the mutated output statement invokes the function,

```
friend ostream& operator << ( ostream& os, my_manipulator object )
```

which actually sets the format for the output's appearance and returns the reference to `cout` so that the item that immediately follows it will be printed in the desired format. After printing one item, format specification will immediately revert to the default.

17.8 Stream Operators with User-Defined Classes

The elegance of streams is that, it can, not only be used with built-in C++ data types, but also with user-defined classes. It requires overloading of the stream insertion and extraction operators. In case of the overloaded friend stream operator << function, the `ostream &` is considered as the first argument. The return value of this friend function is of type `ostream&`. Similarly, for overloading the friend stream operator >> function, the `istream&` is considered as the first argument. The value returned by this friend function is of type `istream&`. In both the cases, a reference to an object of the class to which this operator function is a friend is taken as the second argument. After processing the data members of the second argument, the first argument `istream` object would be returned. Overloading of stream operators to support user-defined data types has been discussed earlier in detail in the chapter on *Operator Overloading*.

The insertion operator, << has been overloaded to have an instance of `ostream` (or one of its derived classes) on the left and an instance of any basic variable type on the right. Similarly, the >> operator is overloaded to have an instance of `istream` class on the left and any basic variable type on the right.

Insertion Operator << Overloading

Consider the prototype of the overloaded << operator to gain a better understanding of streams computation. For instance, the prototype of insertion operator overloaded to display integer data is as follows:

```
ostream & operator << (ostream&, int);
```

Recall that, effectively `cout` is an instance of class `ostream`. Hence, if the variable `num` is an integer, then, the statement

```
cout << num;
```

invokes the overloaded operator function with a reference to `cout` as the first parameter, and the value of the variable `num` as the second. For further overloading, i.e., for this operator to work with user-defined classes, another overloaded function is necessary, similar to the above function declaration. A new operator function accepts a reference to the instance of user-defined class instead of an integer.

Extraction Operator >> Overloading

The >> operator (used with `istream`) can also be overloaded to take care of user-defined types. Inclusion of a function to overload the >> operator helps in writing more compact and readable code in the `main()`. The program `point.cpp` illustrates the overloading of stream operators to operate on user defined data items.

```
// point.cpp: use of both << and >> with a user-defined class.
#include <iostream.h>
// user defined class
class POINT
{
```

```
  private:
     int x, y;
  public:
     POINT()
     {
        x = y = 0;
     }
     friend ostream & operator << ( ostream &os, POINT &p );
     friend istream & operator >> ( istream & is, POINT &p );
};
// friend function to POINT
ostream & operator << ( ostream& os, POINT &p )
{
   os << '(' << p.x << ',' << p.y << ')';
   return os;
}
istream & operator >> ( istream &is, POINT &p )
{
  is >> p.x >> p.y;
  return is;
}

void main()
{
   POINT p1, p2;
   cout << "Enter two coordinate points (p1, p2): ";
   cin >> p1 >> p2;     // invokes overloaded operator >> ()
   cout << "Coordinate points you entered are: " << endl;
   cout << p1 << endl << p2 << endl;   // calls overloaded operator << ()
}
```

Run

```
Enter two coordinate points (p1, p2): 2 3 5 6
Coordinate points you entered are:
(2,3)
(5,6)
```

In `main()`, the statement

```
cin >> p1 >> p2;      // invokes overloaded operator >> ()
```

illustrates cascading of stream operators to read data; the leftmost >> is executed first, and invokes the overloaded operator function with the first parameter as a reference to `cin`, and the second parameter as a reference to the instance of `POINT p1`. The return value of this function (which is `cin` itself) is used as the left hand side of the second >> operator and so on.

The friend function of the class `POINT`,

```
istream & operator >> ( istream &is, POINT &p )
```

overloads the >> operator. It is similar to overloading the output operator. Again, note that the return value enables cascading of the >> operator.

Necessity of Friend Functions

The function overloading the operators >> and << need not always be declared as friend. If the data members x and y were public members of the class POINT, or, if a public member function existed in POINT which output the values of x and y, the friend function declarations would be unnecessary inside the class.

How do the manipulators work with the << operator?

Consider the usage of the manipulator endl:

```
cout << endl;
```

in the previous examples, to insert a newline. The manipulator endl is the function that is declared as,

```
ostream far & endl(ostream far &);
```

in the header file, iostream.h. Thus, endl, is a function that accepts a reference to an ostream (such as cout) and returns the same (a reference to an ostream). Recall that invocation of a function with its name without any parentheses is considered as a pointer to a function. Now it is simple to understand the appearance of the endl on the right side of the << operator; the operator is overloaded to have pointers to functions of this type (that accept a reference to an ostream and returns the same).

Review Questions

17.1 What are streams ? Explain the features of C++ stream I/O with C's I/O system.

17.2 List C++ predefined streams and explain them with suitable example programs.

17.3 Draw console stream class hierarchy and explain its members.

17.4 What is the difference between the statements ?

```
cin >> ch;
ch = cin.get();
```

17.5 Write a program to illustrate the difference between cin and getline while reading strings.

17.6 What is the output of the following statements:

```
(a)  cout << 65;
(b)  cout.put( 65 );
(c)  cout.put( 'A' );
```

17.7 Write a program to print the ASCII table using streams.

17.8 Write an interactive program to print a string entered in a pyramid form. For instance, the string "object" has to be displayed as follows:

```
     o
    o b
   o b j
  o b j e
 o b j e c
o b j e c t
```

17.9 Write an interactive program to print a rectangle with diamond shape gap exactly at the centre of that rectangle. Accept string from standard input device and print output on standard output device. Here is the sample output when the string "object-object" is entered by the user:

```
object-object
object object
objec  bject
obj     ject
ob        ct
o          t
ob        ct
obj      ect
obje    ject
objec  bject
object object
object-object
```

17.10 Write an interactive program to print the salary-slip in the following format:

```
                 Centre for Development of Advanced Computing
                         Bangalore, India - 560 025
                    Salary-Slip for the Month of XXXXXX 1996
--------------------------------------------------------------------
                                                       Date: dd/mm/yy
Employee Name:  xxxxxxxxxxx                         Employee No.:  xxx
Grade: xx                                       Basic Salary: xxxxx.xx
No. of days present: xx

<----PAYMENTS------>     <----DEDUCTIONS----->  <---- RECOVERIES---->
BASIC       xxxxx.xx     PF              xxx.xx  LIC               xxx.x
DA          xxxxx.xx     FPF              xx.xx  CCUBE CONTR.       xx.x
HRA          xxxx.xx     VPF              xx.xx  SOCIETY ADV         x.x
CCA           xxx.xx     BEFUND            x.xx  RENT RECV         xxx.x
DDA             x.xx     P.TAX           xxx.xx  PF LOAN           xxx.x
ARREARS         x.xx     CANTEEN         xxx.xx  SALARY ADV       xxxx.x
ADHOC.ALW     xxx.xx     WELFARE           xx.x  TOUR ADV          xxx.x

TOTAL PAY   xxxxx.xx     TOTAL DED       xxxx.x  TOTAL RECV       xxxx.x
NET PAY: xxxxx.xx
                                                              (SIGNATURE)
```

17.11 Explain the various methods of performing formated stream I/O operations.

17.12 What are manipulators ? List the various predefined manipulators supported by C++ I/O streams.

17.13 How are the input and output streams tied using `istream.tie()` member function ?

17.14 Write a program to display numbers 0 to 10 in octal, decimal, and hexadecimal systems.

17.15 What are custom manipulators ? Write a custom manipulator for inserting 8 spaces in output.

17.16 Explain how standard manipulators are implemented.

17.17 Illustrate parameterized custom manipulators using a suitable program.

17.18 Write a program to overload stream operators for reading and displaying the object of a class `Employee`. The members of this class include `name`, `emp_no`, `DateOfBirth`, `basic`, `grade`, `qualification`, etc.

18

Streams Computation with Files

18.1 Introduction

A computer system stores programs and data in secondary storage in the form of files. Storing programs and data permanently in main memory is not preferred due to the following reasons:

- Main memory is usually too small to permanently store all the needed programs and data.
- Main memory is a volatile storage device, which loses its contents when power is turned off.

The most visible entity in a computer system is a file. The operating system implements the abstract concept of a file by providing file services and managing mass storage devices such as floppy disks, tapes, and hard disks. The various components involved in file processing are shown in Figure 18.1.

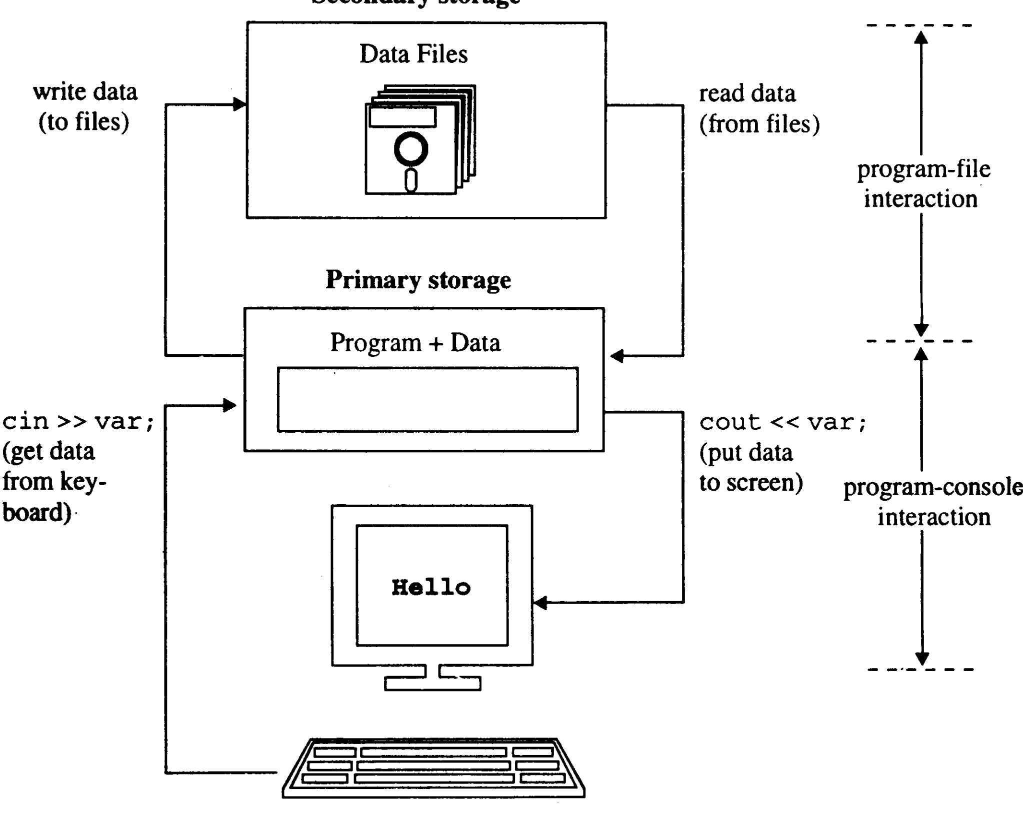

Figure 18.1: Program-console and file interaction

What is a File ?

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data may be numeric, alphabetic, or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by its creator and user. A file is named and is referred to by its name. To define a file properly, it is necessary to consider the operations which can be performed on files. The operating system provides most of the essential file manipulation services such as create, open, write, read, rewind, close, and delete.

A program typically involves data communication between the console and the program or between the files and program, or even both. The program must atleast perform data exchange between processor and main memory. Note that a program without the capability to communicate with the external world will serve no useful purpose (irrespective of the objective with which it is designed).

The streams computation model for manipulating files resemble the console streams model. It uses file streams as a means of communication between the programs and the data files. The input stream supplies data to the program and the output stream receives data from the program. Thus, the *input stream* extracts the data from the file and supplies it to the program, whereas *output stream* stores the data into the file supplied by the program. The movement of data between the disk files and input/output stream in a program is depicted in Figure 18.2.

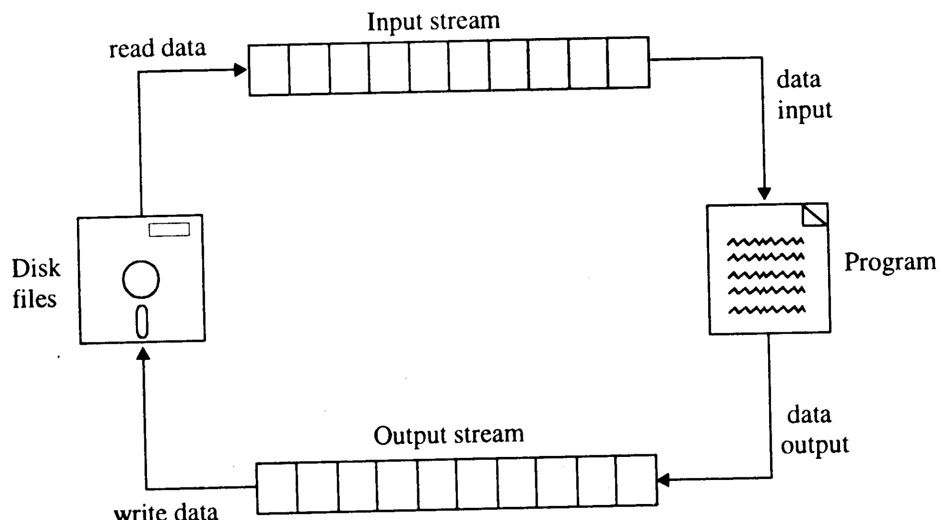

Figure 18.2: File input and output streams

18.2 Hierarchy of File Stream Classes

The file handling techniques of C++ support file manipulation in the form of stream objects. The stream objects `cin` and `cout` are used extensively to deal with the standard input and output devices. These objects are predefined in the header file, `iostream.h` as a part of the C++ language. There are no such predefined objects for disk files. All class declarations have to be done explicitly in the program.

There are three classes for handling files:

- `ifstream` - for handling input files.
- `ofstream` - for handling output files.
- `fstream` - for handling files on which both input and output can be performed.

These classes are derived from `fstreambase` and from those declared in the header file `iostream.h` (`istream`, `iostream`, `ostream`). The hierarchy of C++ file stream classes is shown in Figure 18.3.

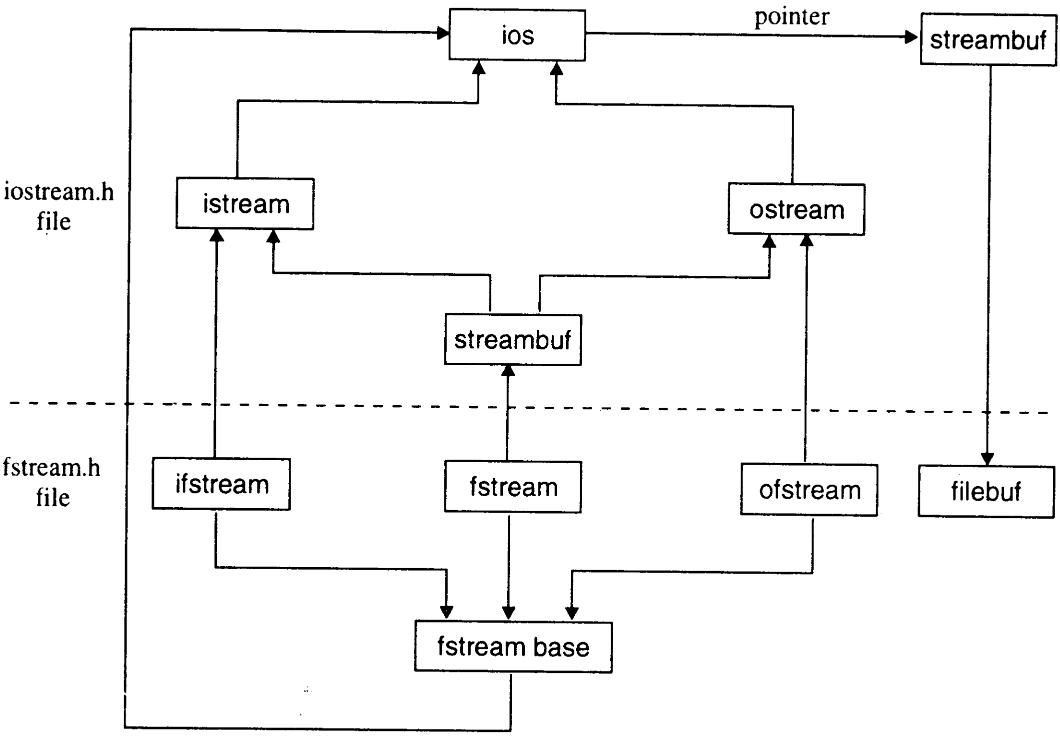

Figure 18.3: Hierarchy of file stream classes

The classes `ifstream`, `ofstream`, and `fstream` are designed exclusively to manage the disk files and their declaration exists in the header file `fstream.h`. To use these classes, include the following statement in the program

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

The actions performed by classes related to file management are described below:

filebuf: The class `filebuf` sets the file buffer to read and write. It contains constant `openprot` used in `open()` of file stream class. It also contains `close()` as a member.

fstreambuf: The class `fstreambuf` supports operations common to the file streams. It serves as a base class for the derived classes `ifstream`, `ofstream`, and `fstream` and contains `open()` and `close()` as member functions.

ifstream: The class `ifstream` supports input operations. It contains `open()` with default input mode and inherits `get()`, `getline()`, `read()`, `seekg()`, and `tellg()` functions from `istream`.

ofstream: The class `ofstream` supports output operations. It contains `open()` with default output mode and inherits `put()`, `seekp()`, `tellp()`, and `write()` functions from `ostream`.

fstream: The class `fstream` supports simultaneous input and output operations. It contains `open()` with default input mode and inherits all the functions from `istream` and `ostream` classes through `iostream`.

18.3 Opening and Closing of Files

In order to process a file, first, it must be opened to get a handle. The file handle serves as a pointer to the file. Typically, manipulation of a file involves the following steps:

- Name the file on the disk
- Open the file to get the file pointer
- Process the file (read/write)
- Check for errors while processing
- Close the file after its complete usage

The filename is a string of characters, with which a file is logically identified by the user. It provides a means to communicate with the user transparently. The number and type of characters used in naming a file depends on the operating system. Normally, a file has two parts: a *primary name* and an *optional extension*. If the file name has an extension, it is separated by a period from the primary name. Some of the valid file names in the MS-DOS based machines are the following:

```
student.cpp
data.txt
copy.exe
student.obj
student.exe
TEMP
data1
tax.in
```

In MS-DOS systems, the maximum size of a primary name is eight characters and that of an extension is three characters. However, in UNIX based machines, the file name can be upto 31 characters and any number of extensions separated by dots. Some valid file names in the UNIX system include all those valid in the MS-DOS and in addition, it includes the following:

`.login` (no primary name, acts as hidden file)
`xyz.txt.mine`
`text_data_file`
`student.8sem.raj`

In order to get a file pointer, first the file must be created (if it does not exist) and linked to the file name. A file stream can be defined using stream classes, `ifstream`, `ofstream`, or `fstream` depending on the purpose (read or write). In C++, a file can be opened using the following:

- The constructor function of the class.
- The member function `open()` of the class.

After processing an opened file, it must be closed. It can be closed either by explicitly using the `close()` member function of the class or it is automatically closed by the destructor of the class, when the file stream object goes out of scope (expires).

Opening Files Using Constructors

In order to access a file, it has to be opened either in read, write, or append mode. In all the three file stream classes, a file can be opened by passing a filename as the first parameter in the constructor itself. For example, the statement

```
ifstream infile("test.txt");
```

opens the file `test.txt` for input. It is known that, a constructor is used to initialize an object during its creation. Hence, the constructor can be utilized to initialize the filename to be used with the file stream object. The creation and assignment of file name to the file stream object involves the following steps:

- Create a file stream object using the appropriate class depending on the type of file stream required. For example, `ifstream` can be used to create the input stream, `ofstream` can be used to create the output stream, and `fstream` can be used to create the input and output stream.
- Bind the file stream to the disk. In disk, file stream is identified by a file name.

For instance, the following statement opens a file named `database` for input:

```
ifstream infile ( "database" );
```

It creates `infile` as the object of the class `ifstream` that manages the input stream, and opens the file `database` and binds it to the output stream disk file. Similarly, the statement

```
ofstream outfile( "data.out" );
```

defines `outfile` as the object of the class `ostream`, and binds it to the file `data.out` for writing.

The program statements can refer to the file objects similar to the stream objects. The syntax for performing I/O operations with standard input-output devices also holds good for files. For instance, to print the message `Hello World` on the console and into the file, the following commands can be issued:

```
cout << "Hello World";
```

prints the message `Hello World` on the standard output device. Whereas, the statement

```
myfile << "Hello World";
```

prints the message `Hello World` into the file pointed to by the file pointer `myfile` (Figure 18.4).

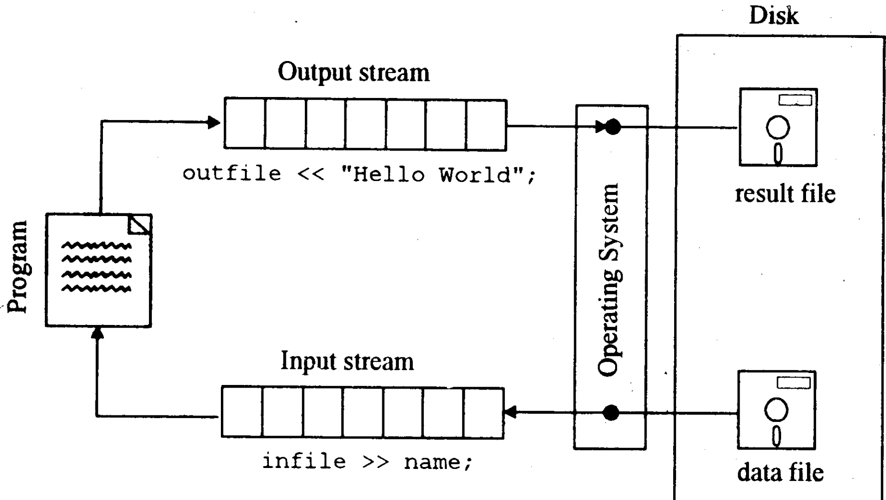

Figure 18.4: File I/O with stream operators

The following statements:

```
outfile << "Hello World";   // write string constant
outfile << salary;          // write variable content
outfile << 750;                         // write 750 to file
```

prints the string `"Hello World"` and the contents of the variable `salary` to the output file. Similarly, the following statements:

```
infile >> name;                         // read string
infile >> age;                          // read integer
infile >> number;           // read float
```

read the variables `name`, `age`, and `number` from the input file stream `infile`.

The constructors of all these classes are declared in the header file `fstream.h`. The prototypes of file stream constructors are shown in Figure 18.5.

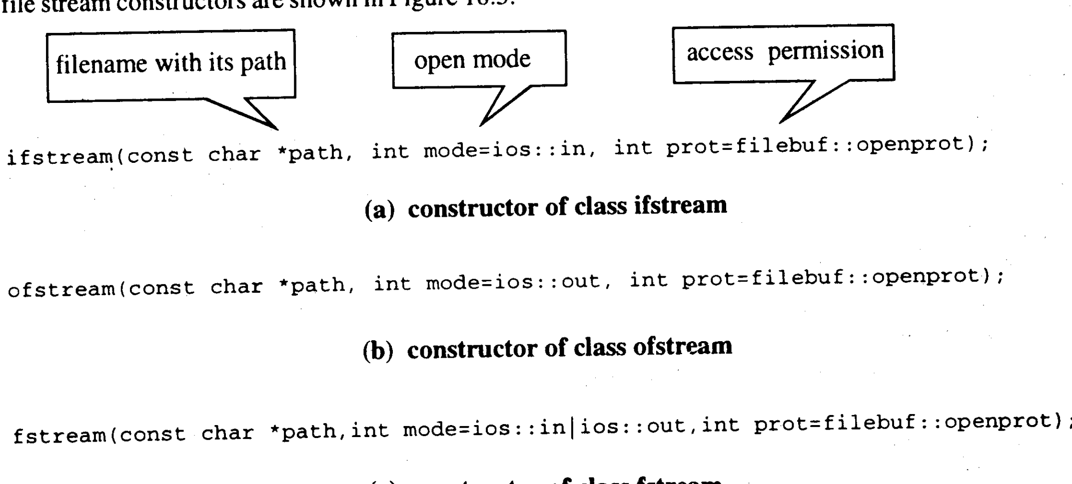

Figure 18.5: Prototype of file stream class constructors

The stream class arguments have the following meaning:

`path`: It specifies the pathname of the file to be opened. If the file is in the current directory, only the filename needs to be specified. Otherwise, separate the directory names by a backslash (\) in the MS-DOS or a slash (/) in the Unix operating systems.

`mode`: It specifies the mode in which the file is to be opened. The argument may be specified by using enumerated constants declared in the `ios` class.

`prot`: It specifies the access permission. It is not used if `ios::nocreate` is used in *mode*. The default permissions are set in the static variable `filebuf::openprot` for both read and write (The file can be read from and written to) permissions. The access permissions can be read only (`S_IREAD`) or write only (`S_IWRITE`). Under UNIX, `prot` parameter can be used to specify read, write, and execute permissions to specific owner categories (viz., user, group and others).

The file must be closed to release all the resources allocated to it. It is known that, the destructor normally does the cleanup operation. Whenever file stream object goes out of scope or the program

terminates its execution, the file is automatically closed by destructor. The program `stdfile.cpp` creates a file `student.out` using constructors and writes student details into it.

```
// stdfile.cpp: student file, creating file with constructor function
#include <fstream.h>
void main()
{
   char name[30];
   int marks;
   ofstream fout ( "student.out" ); // connect student.out to fout
   // read first student details
   cout << "Enter Name: ";
   cin >> name;
   cout << "Enter Marks Secured: ";
   cin >> marks;
   // write to a file
   fout << name << endl;
   fout << marks << endl;
   // read second student details
   cout << "Enter Name: ";
   cin >> name;
   cout << "Enter Marks Secured: ";
   cin >> marks;
   // write to a file
   fout << name << endl;
   fout << marks << endl;
}
```

Run

```
Enter Name: Rajkumar
Enter Marks Secured: 95
Enter Name: Tejaswi
Enter Marks Secured: 90
```

Note: On execution the file `student.out` contains the following.

```
Rajkumar
95
Tejaswi
90
```

In `main()`, the statement

```
ofstream fout ( "student.out" ); // connect student.out to fout
```

creates the object `fout` and binds it to the file `student.out` by opening it in the write mode. The statement

```
fout << name << endl;
```

writes the string name to the file, and the statement

```
fout << marks << endl;
```

writes the integer variable `marks` to the file. The file `student.out` is closed automatically when the program terminates.

Note that, when a file is opened in write-only mode, a new file is created if a file with the same name does not exists. Otherwise, the current contents of the file is truncated and opened in write mode. The program `stdread.cpp` opens file `student.out` using a constructor and prints its contents on the console.

```
// stdread.cpp: student file, read the file student.out
#include <fstream.h>
void main()
{
   char name[30];
   int marks;
   ifstream fin ( "student.out" );  // connect student.out to fout
   // read first student details
   fin >> name;
   fin >> marks;
   cout << "Name: " << name << endl;
   cout << "Marks Secured: " << marks << endl;
   // read second student details
   fin >> name;
   fin >> marks;
   cout << "Name: " << name << endl;
   cout << "Marks Secured: " << marks << endl;
}
```

Run

```
Name: Rajkumar
Marks Secured: 95
Name: Tejaswi
Marks Secured: 90
```

The above program must be executed only when a file with the name `student.out` already exists and has data as expected by the program.

Opening and Closing of Files Explicitly

The file can also be opened explicitly using the function `open()` instead of a constructor. This mechanism is generally used when different files are to be associated with the same object at different times. The syntax for opening a file is shown in Figure 18.6. The file can be closed explicitly using the `close()` function as follows:

```
stream_object.close();
```

The following examples illustrates file open and close operations.

1. Opening file in write mode:

```
ofstream fout;                          // create stream for output
....
fout.open( "student.out" ); // bind stream to file
....
fout.close();                    // disconnect stream from student.out
...
```

```
fout.open( "person.out" );          // bind stream to another file
....
```

2. Opening file in read mode:

```
ofstream fin;                       // create stream for input
....
fin.open( "student.in" );           // bind stream to file
....
fin.close();                        // disconnect stream from student.in
...
fin.open( "student.out" );          // bind stream to another file
....
```

There is a limit on the maximum number of files which can be opened. This constraint is imposed by the underlying operating system on which a program executes. For instance, in MS-DOS, the entry `FILES=N` in the `CONFIG.SYS` file; the entry `FILES = 20` indicates there can be a maximum of 20 files opened at a time. If any attempt is made to open a file above this limit, it fails and returns the NULL handle. Therefore, it is advisable to close a file when it is no longer needed.

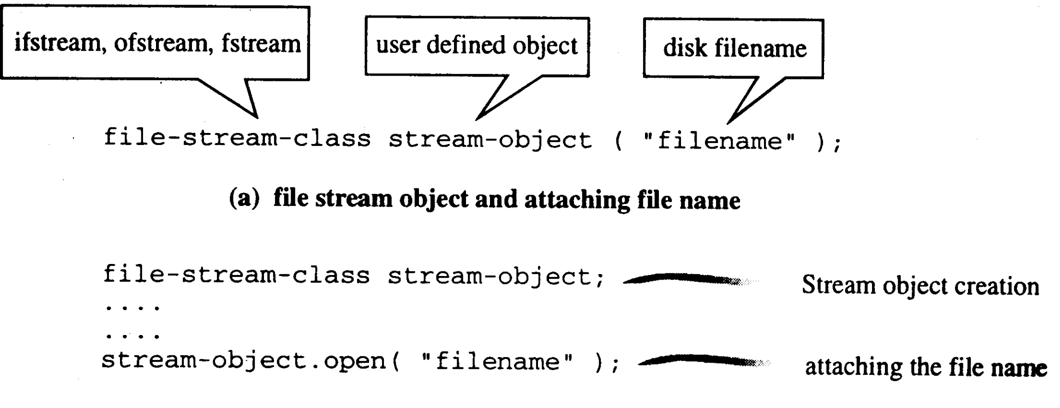

Figure 18.6: Syntax of opening the file

18.4 Testing for Errors

The assumption of a file operation (opening, processing, or closing) is always successful in an ideal situation. There are situations, when the user tries to open a non-existent file in read-mode or tries to open a file in write mode which has been marked as read-only. File operations fail under such circumstances. Such errors must be trapped and appropriate actions must be taken before further processing.

This can be done using the operator `!` with an instance of the `ifstream`, `ofstream` or `fstream`. The operator `!` is overloaded to return nonzero in case any stream errors have occurred. For example, to open a file for input and test whether it has successfully opened (it will not be opened if the file does not exist), the following code may be used:

```
ifstream in_file( "test.txt" );
//test for error
if( !in_file )
{  //File wasn't opened
   cerr << "Cannot open test.txt\n";
   exit( 1 );
}
```

Once the file has been opened successfully, a common activity is to read from the file while the end-of-file has not yet been reached. Using the name of a file stream instance in place of a condition expression (such as inside an `if` or `while` statement) evaluates to nonzero only when no errors have occurred in the file. Hence, errors such as end-of-file can be tested as follows:

```
while( in_file )    // while EOF has not been reached
{
   //Read from the file.
}
```

where `in_file` is an instance of `ifstream`, but an instance of `ofstream` or `fstream` can equally be used in such situations.

An example using `ifstream` to output the contents of a file is given below. Note that, the use of the manipulator `resetiosflags` to prevent skipping white-space characters in the input. A program to display the contents of a file (filename is entered interactively) on the console is listed in `fdisp.cpp`.

```
// fdisp.cpp: display file contents using ifstream to input from a file
#include <fstream.h>
#include <iomanip.h>
int main()
{
   char ch;
   char filename[ 25 ];
   cout << "Enter Name of the File: ";
   cin >> filename;
   // create a file object in read mode
   ifstream ifile( filename );
   if( !ifile )   // file open status
   {
      cerr << "Error opening " << filename << endl;
      return 1;
   }
   ifile >> resetiosflags( ios::skipws ); // do not skip space or new line
   //Comment above line; then execute the program, you will see funny result
   while( ifile )   // while EOF not reached.
   {
      ifile >> ch;   // read a character from file
      cout << ch;    // display character on console
   }
   return 0;
}
```

Run

```
Enter Name of the File: mytype.cpp
```
[The contents of the input file, mytype.cpp is displayed on console]

In main(), the statement

```
ifstream ifile( filename );
```

creates the disk file object, ifile for a file name entered interactively in the read mode. In the absence of the statement,

```
ifile >> resetiosflags( ios::skipws );
```

the file will be displayed without any spaces or newlines, since the >> operator, neglects any white-space characters by default. The statement

```
ifile >> ch;
```

reads a character from the file in a manner similar to cin. It does not skip white-space characters since ios::skipws flag is reset. The object ifile becomes 0 as soon as it reaches the end of the file and hence, the statement

```
while( ifile )
```

loops until end of file is reached. All those files that are opened by a program must be closed by it. Otherwise, the system closes all those files which are in open state during the termination of a program.

The program keyin.cpp waits for keyboard input and dumps all input characters into the file key.txt until the end-of-file (Ctrl-Z) character is pressed followed by the carriage-return key.

```
// keyin.cpp: Reads all the characters entered and stores the same in the file
#include <fstream.h>
void main()
{
   char ch;
   cout<<"Enter characters..<Ctrl-Z followed by carriage-return to stop>\n";
   ofstream ofile( "key.txt" ); // opens file in output ASCII mode
   while( cin )    // not end of file
   {
      cin.get( ch ); // read character from console
      ofile << ch;   // write to file
   }
   ofile.close(); // close file
}
```

Run

```
Enter characters..<Ctrl-Z followed by carriage-return to stop>
1
A B C .. X Y Z
^Z
```

Note: The file key.txt has all the above characters except ^Z

In main, the statement

```
ofstream ofile( "key.txt" );
```

opens the file key.txt in output mode. The statement

```
cin.get( ch );
```

reads a character from the input device without skipping white-space characters. Hence, the `resetiosflags(ios::skipws)` manipulator need not be used to prevent skipping of white-space characters. The statement

```
ofile << ch;
```

writes character to the output file. The statement

```
ofile.close();
```

closes the file.

Another approach for detecting the end-of-file condition is using the member function `eof()`. This operates as follows:

```
stream-object.eof() = 0 if end-of-file is not detected
                    = non-zero if end-of-file is detected
```

The function `eof()` is a member function of the class `ios`. For example

```
if( fin.eof() )
   // end-of-file
else
   // not end-of-file
```

The program `stdwr.cpp` illustrates the processing of errors that occur while manipulating files.

```
// stdwr.cpp: student file, creating, writing, and reading the same
#include <fstream.h>
void student_write( int count )
{
   char name[30];
   int i, marks;
   // create a file, open it in write mode and save data
   ofstream fout;    // create a file object
   fout.open( "student.out" );   // connect file object to file
   if( !fout )
   {
      cout << "Error: " << "student.out cannot be opened in write mode";
      return;
   }
   for( i = 0; i < count; i++ )
   {
      cout << "Enter Name: ";
      cin >> name;
      cout << "Enter Marks Secured: ";
      cin >> marks;
      // write to a file
      fout << name << endl;
      fout << marks << endl;
   }
   fout.close();  // disconnect a file
}
```

```
void student_read()
{
   char name[30];
   int i, marks;
   // create a file, open it in write mode and save data
   ifstream fin;     // create a file object
   fin.open( "student.out" ); // connect file object to file
   if( !fin )
   {
      cout << "Error: " << "student.out cannot be opened in read mode";
      return;
   }
   while(1)
   {
      fin >> name;
      fin >> marks;
      if( fin.eof() )
         break;
      cout << "Name: " << name << endl;
      cout << "Marks Secured: " << marks << endl;
   }
   fin.close();   // disconnect a file
}
void main()
{
   int count;
   cout << "How many students ? ";
   cin >> count;
   cout << "Enter student details to be stored..." << endl;
   student_write( count );
   cout << "Student details processed from the file..." << endl;
   student_read();
}
```

Run

```
How many students ? 3
Enter student details to be stored...
Enter Name: Mangala
Enter Marks Secured: 75
Enter Name: Chatterjee
Enter Marks Secured: 99
Enter Name: Rao-M-G
Enter Marks Secured: 50
Student details processed from the file...
Name: Mangala
Marks Secured: 75
Name: Chatterjee
Marks Secured: 99
Name: Rao-M-G
Marks Secured: 50
```

In `student_write()`, the statement

```
fout.open( "student.out" );    // connect file object to file
```

opens the file `student.out` and connects the same to the stream object `fout`. The statement

```
if( !fout )
```

verifies whether the file is opened successfully or not. If condition is true, when `!fout` is nonzero.

The statement in `student_read()`

```
if( fin.eof() )
   break;
```

checks for the end-of-file and terminates file processing if the end-of-file is reached.

18.5 File Modes

The constructors of `ifstream` and `key.txt` and the function `open()` are used to create files as well as open the existing files in the default mode (text mode). In both methods, the only argument used is the filename. C++ provides a mechanism of opening a file in different modes in which case the second parameter must be explicitly passed. The syntax is as follows:

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

It opens the file in the specified mode. The list of file modes are shown in Table 18.1 with mode value and their meaning.

mode value	Effect on the mode
`ios::in`	open for reading.
`ios::out`	open for writing.
`ios::ate`	seek (go) to the end of file at opening time.
`ios::app`	append mode: all writes occur at end of file.
`ios::trunc`	truncate the file if it already exists.
`ios::nocreate`	open fails if file does not exist.
`ios::noreplace`	open fails if file already exists.
`ios::binary`	open as a binary file.

Table 18.1: File open modes

The following points can be noted regarding file modes:

- Opening a file in `ios::out` mode also opens it in the `ios::trunc` mode by default. That is, if the file already exists, it is truncated.
- Both `ios::app` and `ios::ate` sets pointers to the end-of-file, but they differ in terms of the types of operations permitted on a file. The `ios::app` allows to add data from the end-of-file, whereas `ios::ate` mode allows to add or modify the existing data anywhere in the file. In both the cases, a file is created if it is non-existent.
- The mode `ios::app` can be used only with output files.
- The stream classes `ifstream` and `ofstream` open files in read and write modes respectively by default.

- For `fstream` class, the mode parameter must be explicitly passed.
- More than one value may be ORed to have a combined effect. For instance, the following statement opens a file for reading in binary mode:

```
istream in_file( "myfile", ios::in | ios::binary );
```

The program `payfile.cpp` generates a payroll-like output and directs the output to the file `pay.txt` instead of `cout`. It stores floating point data in the form of ASCII characters instead of machine representation (binary form).

```
// payfile: payroll like output example printing results to file
#include <fstream.h>
#include <iomanip.h>
void main()
{
  float f1=123.45, f2=34.65, f3=56;
  // open file "pay.txt" in output mode and truncate its contents if exists
  ofstream out_file( "pay.txt", ios::trunc );
  out_file << setiosflags(ios::showpoint|ios::fixed)
       << setiosflags(ios::right);
  out_file << setw(6) << f1 << endl;
  out_file << setw(6) << f2 << endl;
  out_file << setw(6) << f3 << endl;
}
```

Run

After execution of the program, the file `pay.txt` contains the following:

```
123.45
 34.65
 56.00
```

In `main()`, the statement

```
ofstream out_file( "pay.txt", ios::trunc );
```

creates the file `pay.txt` and truncates its contents if the file already exists. As with the console streams, manipulators can be used with any of the file stream instances.

18.6 File Pointers and their Manipulations

The knowledge of the logical location at which the current read or write operations occur is of great importance in achieving faster access to information stored in a file. The file management system associates two pointers with each file, called *file pointers*. In C++, they are called *get pointer* (input pointer) and *put pointer* (output pointer). These pointers facilitate the movement across the file while reading or writing. The *get pointer* specifies a location from where the current reading operation is initiated. The *put pointer* specifies a location from where the current writing operation is initiated. On completion of a read or write operation, the appropriate pointer will be advanced automatically.

Default Actions

The file pointers are set to a suitable location initially based on the mode in which the file is opened. Fundamentally, a file can be opened in the read mode, write mode, or append mode. The logical location of file pointers when a file is opened is discussed below (see Figure 18.7.):

- **Read-only Mode:** when a file is opened in read-only mode, the input (get) pointer is initialized to point to the beginning of the file, so that the file can be read from the start.
- **Write-only Mode:** when a file is opened in write-only mode, the existing contents of the file are deleted (if a given file already exists) and the output pointer is set to point to the beginning of the file, so that data can be written from the start.
- **Append Mode:** when a file is opened in append mode, the existing contents of the file remain unaffected (if a given file already exists) and the output pointer is set to point to the end of the file so that data can be written (appended) at the end of the existing contents.

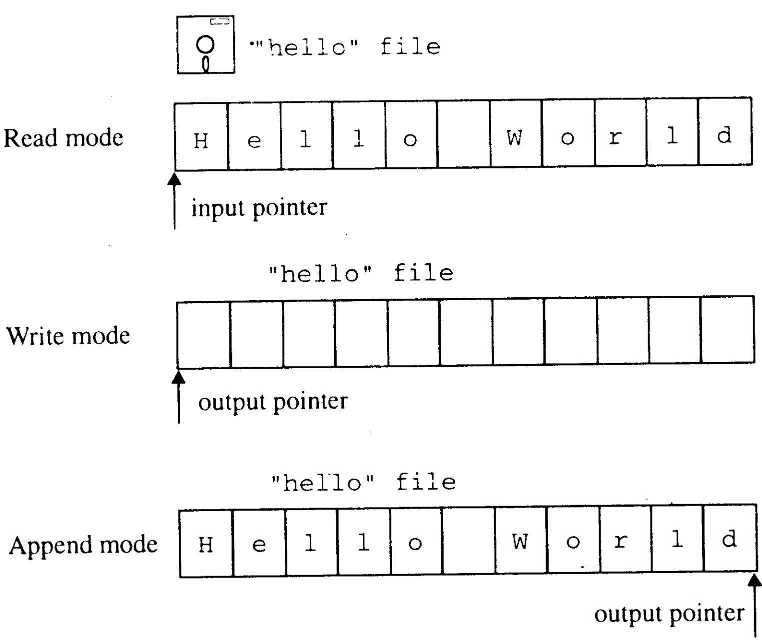

Figure 18.7: File pointer position on opening a file

Functions for Manipulation of File Pointers

The C++ I/O system supports four functions for setting a file pointer to any desired position inside the file or to get the current file pointer. These allow the programmer to have control over a position in the file where the read or write operation takes place. The functions are listed in the Table 18.2.

Function	Member of the class	Action Performed
`seekg()`	`ifstream`	Moves get file pointer to a specific location
`seekp()`	`ofstream`	Moves put file pointer to a specific location
`tellg()`	`ifstream`	Returns the current position of the get pointer
`tellp()`	`ofstream`	Returns the current position of the put pointer

Table 18.2: File pointer control functions

The `seekp()` and `tellp()` are member functions of `ofstream`. The `seekg` and `tellg` are member functions of `ifstream`. The class `fstream` deals with files in both input and output modes. Hence, there are two file pointers in class `fstream` - the *put pointer* used for writing and the *get pointer* used for reading. All four functions mentioned above are available in the class `fstream`. The `seekp()` and `tellp()` deal with the `put` pointer, while `seekg()` and `tellg()` deal with the `get` pointer.

The two `seek` functions have the following prototypes:

```
istream & seekg(long offset, seek_dir origin = ios::beg);
ostream & seekp(long offset, seek_dir origin = ios::beg);
```

Both functions set a file pointer to a certain offset relative to the specified origin. The second parameter `origin`, represents the reference point from where the offset is measured. It can be specified by using an enumeration declaration `(seek_dir)` given in the `ios` class. (See Table 18.3.)

origin value	Seeks from...
`ios::beg`	seek from beginning of file
`ios::cur`	seek from current location
`ios::end`	seek from end of file

Table 18.3: File seek origins

For example, the statement

```
infile.seekg( 20, ios::beg );
or
infile.seekg( 20 );
```

moves the file pointer to the 20th byte in the file, `infile`. After this, if a read operation is initiated, the reading starts from the 21st item (bytes in file are numbered from zero) within the file. The statement

```
outfile.seekp( 20, ios::beg );
or
outfile.seekp( 20 );
```

moves the file pointer to the 20th byte in the file `outfile`. After this, if write operation is initiated, the writing starts from the 21st item (bytes in file are numbered from zero) within the file. Consider the following statements:

```
ofstream outfile( "student.out", ios::app );
int size = outfile.tellp();
```

The first statement creates the file stream object `outfile`, and connects it to the disk file, `student.out`. It moves the output pointer to the end of the file. The second statement assigns the value of the *put pointer* to the integer variable `size`, which in this case represents the number of bytes in the file. The program `fsize.cpp` prints the size of a file, whose name is given as a command line parameter.

```
// fsize.cpp: file size finding using seekg and tellg
#include <fstream.h>
int main( int argc, char *argv[] )
{
   if( argc < 2 ) // no filename is passed
```

```
    {
       cout << "Usage: fsize <filename>";
       return 1;
    }
    ifstream infile( argv[ 1 ] ); // file open in read and write mode
    if( !infile )  // open success
    {
       cerr << "Error opening " << argv[ 1 ] << endl;
       return 1;
    }
    infile.seekg( 0, ios::end ); // set read pointer to end of file
    cout << "File Size=" << infile.tellg(); // read current position
    return 0;
}
```

Run1

```
Usage: fsize <filename>
```

Run2

```
File Size=437
```

In `main()`, the statement

```
infile.seekg( 0, ios::end );
```

moves the *read pointer* to the end of the file, and the statement

```
infile.tellg();
```

reads the `get` pointer value. In this situation, it represents the size of the file.

The `seekg()` sets the get pointer while `seekp()` sets the put pointer to the specified location. Some of the pointer offset calls and their actions are shown in Table 18.4 and Figure 18.8. It is assumed that the variable `fout` is the object of the stream class `ofstream` and `fin` is the object of the stream class `ifstream`.

Seek call	Action performed
`fout.seekg(0, ios::beg)`	Go to the beginning of the file
`fout.seekg(0, ios::cur)`	Stay at the current file
`fout.seekg(0, ios::end)`	Go to the end of the file
`fout.seekg(n, ios::beg)`	Move to (n+1) byte location in the file
`fout.seekg(n, ios::cur)`	Move forward by n bytes from current position
`fout.seekg(-n, ios::cur)`	Move backward by n bytes from current position
`fout.seekg(-n, ios::end)`	Move backward by n bytes from the end
`fin.seekp(n, ios::beg)`	Move write pointer to (n+1) byte location
`fin.seekp(-n, ios::cur)`	Move write pointer backward by n bytes

Table 18.4: Seek calls and their actions

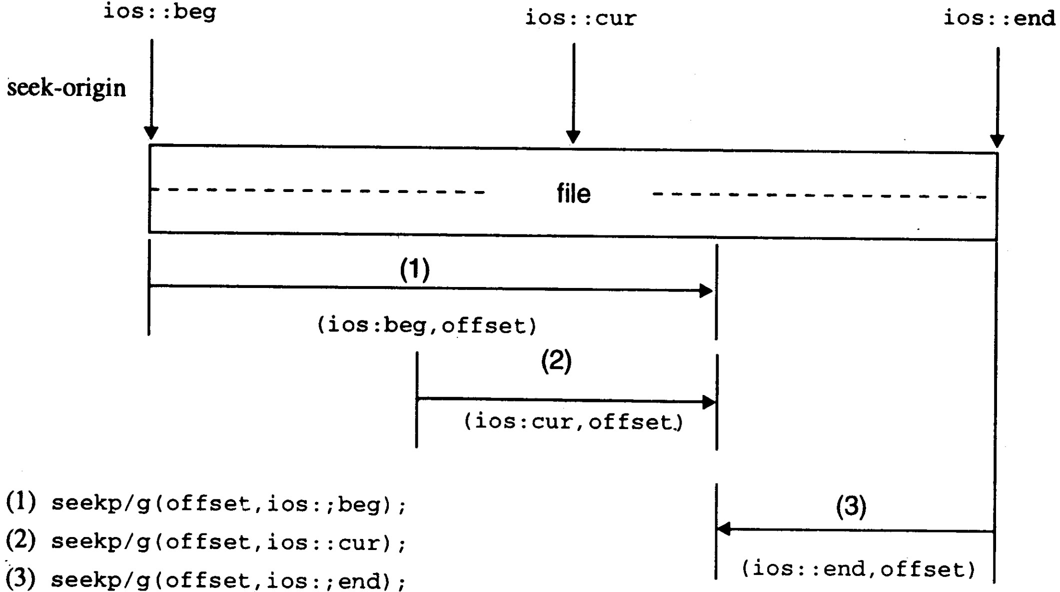

Figure 18.8: Seek positions and their origin

18.7 Sequential Access to a File

Unlike other programming languages (such as COBOL), C++ does not provide commands organizing and processing files as sequential or direct (random) files. However, it provides file manipulation commands which can be used by the programmer to device access to files sequentially or randomly. A *sequential file* has to be accessed sequentially; to access the particular data in the file all the preceding data items have to be read and discarded. A *random file* allows access to the specific data without the need for accessing its preceding data items. However, it can also be accessed sequentially. Organizing a file either as sequential or random depends on the type of *media* on which the file is organized and stored. For instance, a file on a tape must be accessed sequentially, whereas, a file on a hard disk or floppy disk can be accessed either sequentially, or randomly. In C++, it is the responsibility of the programmer to devise a mechanism for accessing a file.

The C++ file stream system supports a wide variety of functions to perform the input-output operation on files. The functions, `put()` and `get()`, are designed to manage a single character at a time. The other functions, `write()` and `read()`, are designed to manipulate blocks of character data.

The put() and get() Functions

The function `get()` is a member function of the file stream class `fstream`, and is used to read a single character from the file. The function `put()` is a member function of the output stream class `fstream`, and is used to write a single character to the output file. The program `putget.cpp` reads a string from the standard input device, and writes the same to a file character by character. A sequential file is created and its pointer is positioned at the beginning of the file. It is processed sequentially until the end-of-file is encountered.

```
// putget.cpp: writes and reads characters from the file
#include <fstream.h>
void main()
{
   char c, string[ 75 ];
   fstream file( "student.txt", ios::in | ios::out );
   cout << "Enter String: ";
   cin.getline( string, 74 );
   for( int i = 0; string[i]; i++ )
      file.put( string[i] );
   file.seekg(0); // seek to the beginning
   cout << "Output String: ";
   while( file )
   {
      file.get( c ); // reads a character
      cout << c;
   }
}
```

Run

```
Enter String: Object-Computing with C++
Output String: Object-Computing with C++
```

Note: The file `student.txt` contains the entered string.

The stream `fstream` provides the facility to open a file in both read and write modes; so that the file can be processed randomly by positioning the file pointers.

18.8 ASCII and Binary Files

The stream operators insertion and extraction always manipulate and deal with formatted data. Data has to be *formatted* to produce logical information. This is because, most of the I/O devices communicate to the computer system using ASCII code, but CPU processes these data using the binary system. Hence, it is necessary to convert data while reading from the input device or displaying data on output device. Most visible data formatting operation is alignment of display fields. In addition to this, data formatting operation also occur transparently while transferring data between the program and console or a file. For example, in order to display an integer value, the << operator converts the number into a stream of ASCII characters. Similarly, the >> operator converts the input ASCII characters to binary while reading data from the input device. For instance, when a number, say, 120 is typed in response to an input statement such as:

```
cin >> i;
```

The user enters data by typing on the keyboard. In this case, stream operator receives ASCII codes of the numeric characters 1, 2, and 0 (which are 49, 50, and 48). The >> operator function converts the input ASCII data to binary and assigns to the variable `i`. Similarly, the << operator in a statement such as:

```
cout << i;
```

converts the content of the variable `i` (say 120) into three ASCII characters, 49, 50, and 48 and then sends the same to the standard output device. The representation of an integer in the character form and binary form is shown in Figure 18.9.

The program `objsave.cpp` illustrates the flexibility gained by overloading the insertion and extraction operators while saving objects into a file or retrieving objects from a file.

```
// objsave.cpp: saving a object to a file with stream operator overloaded
#include <fstream.h>
#include <ctype.h>      // for toupper
#include <string.h>     // for strlen
#define MAXNAME 40
class Person
{
   private:
      char name[ MAXNAME ];
      int age;
   public:
      // this function writes the class's data members to the file
      void write( ofstream &os )
      {
         os.write( name, strlen( name ) );
         os << ends;
         os.write( (char*)&age, sizeof( age ) );
      }
      // this function reads the class's date member from the file.
      // It returns nonzero if no errors were encountered while reading
      int read( ifstream &is )
      {
         is.get( name, MAXNAME, 0 );
         name[ is.gcount() ] = 0;
         is.ignore( 1 );   // ignore the NULL terminator in the file.
         is.read( (char*)&age, sizeof( age ) );
         return is.good();
      }
      // stream operator, << overloading
      friend ostream & operator << ( ostream &os, Person &b );
      // stream operator >> operator overloading
      friend istream &operator >> ( istream &is, Person &b );
      // output file stream operator overloading
      friend ofstream &operator << ( ofstream &fos, Person &b )
      {
         b.write( fos );
         return fos;
      }
      // output file stream operator overloading
      friend ifstream &operator >> ( ifstream &fos, Person &b )
      {
         b.read( fos );
         return fos;
      }
};
istream &operator >> ( istream &is, Person &b )
{
   cout << "Name: ";
```

```
   is >> ws;   // flush input buffer
   is.get( b.name, MAXNAME );
   cout << "Age : ";
   is >> ws >> b.age;
   return is;
}
ostream &operator << ( ostream &os, Person &b )
{
   os << b.name << endl;
   os << b.age << endl;
   return os;
}
void main()
{
   Person p_obj;
   // open a file in binary mode and write objects to it
   ofstream ofile( "person.txt", ios::trunc|ios::binary );
   char ch;
   do
   {
      cin >> p_obj;      // read object
      ofile << p_obj;    // write object to the output file
      cout << "Another ? ";
      cin >> ch;
   } while( toupper( ch ) == 'Y' );
   ofile.close();
   // Output loop, display file content
   ifstream ifile( "person.txt", ios::binary );
   cout << "The objects written to the file were:.." << endl;
   while( 1 )
   {
      ifile >> p_obj;    // extract person object from file
      if( ifile.fail() )    // file read fail, end-of-file
         break;
      cout << p_obj;     // display person object on console
   }
}
```

Run

```
Name: Tejaswi
Age : 5
Another ? y
Name: Savithri
Age : 23
Another ? n
The objects written to the file were:..
Tejaswi
5
Savithri
23
```

In the above program, the object `p_obj` of the class `Person` is retrieved from or saved to a file just like a variable of a built-in data type. The statement

```
cin >> p_obj;
```

reads the object, `p_obj` from the standard input device, whereas, the statement

```
ifile >> p_obj;
```

retrieves the object, `p_obj` from the input file `ifile`. The statement

```
cout << p_obj;
```

displays the object, `p_obj` on the standard output device and the statement

```
ofile << p_obj;
```

stores the object `p_obj` in the file. The mechanism of manipulating user defined objects with stream operators is depicted in Figure 18.10.

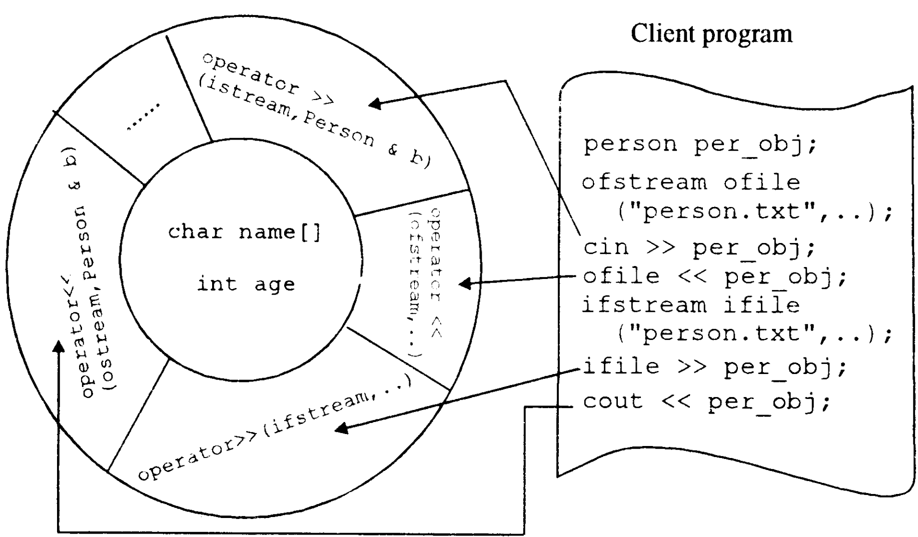

Figure 18.10: Files and objects interaction

The classes `ifstream` and `ofstream` are declared in the `fstream.h` header file. The member functions of the stream classes `ifstream` and `ofstream`, `get()` and `write()` can be used to manipulate user defined objects in disk files. These functions handle the entire structure of an object as a single unit, and store or retrieve in binary format. For instance, the member function `write()` of the class `ofstream`, writes a class's object from memory byte-by-byte without conversion to the target disk file opened in binary mode. It is important to note that, only *data members* of a class are copied to the disk file. For instance, the statement in the above program,

```
ofile << p_obj;
```

can be replaced by the statement,

```
ofile.write( (char *) &p_obj, sizeof(p_obj) );
```

to store the object `p_obj` to the disk file. Likewise, the statement

```
ifile >> p_obj;
```

can be replaced by:

```
ifile.read( (char *) &p_obj, sizeof(p_obj) );
```

in order to retrieve the object from the disk file. The length of the object is computed using the `sizeof` operator. It returns the number of bytes required to hold all the data members of the `p_obj` object.

18.10 File Input/Output with fstream Class

The class `fstream` supports simultaneous input and output operations. It contains `open()` with input mode as default. It inherits all the functions from `istream` and `ostream` classes through `iostream`. The program `student.cpp` illustrates the role of `fstream` class in the manipulation of files. It reads the data from the input file `student.in` and writes the processed information into another disk file `student.out`.

```
// student.cpp: reads students from files and writes result to another file
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
#include <process.h>
void main()
{
    fstream infile;  // input file
    fstream outfile; // output file
    int i, count, percentage;
    char name[30];
    // Open source file for reading
    infile.open( "student.in", ios::in );
    if( infile.fail() )
    {
       cout << "Error: student.in file non-existent";
       exit( 1 );
    }
    outfile.open( "student.out", ios::out );
    if( outfile.fail() )
    {
       cout << "Error: unable to open student.out in write mode";
       exit( 1 );
    }
    infile >> count; // how many students
    // write header to output file
    outfile << "       Students Information Processing" << endl << endl;
    outfile << "-------------------------------------------" << endl;
    for( i = 0; i < count; i++ )
    {
      // read data and percentage secured from the input file
      infile >> name;
      infile >> percentage;
      // write name and class secured based on percentage to output file
      outfile << "Name: " << name << endl;
      outfile << "Percentage: " << percentage << endl;
```

```
        outfile << "Passed in: ";
        if( percentage >= 70 )
           outfile << "First class with distinction";
        else
           if( percentage >= 60 )
              outfile << "First class";
           else
              if( percentage >= 50 )
                 outfile << "Second class";
              else
                 if( percentage >= 35 )
                    outfile << "Third class";
                 else
                    outfile << "Sorry, Failed!";
        outfile << endl;
       outfile << "----------------------------------------------" << endl;
    }
    // close files
    infile.close();
    outfile.close();
}
```

Run

Note that before running the above program, create the input file `student.in` containing the data according to the following format:

1. Number of students
2. First student name (without blanks)
3. First student percentage score

....

....

N. Last student name
   Last student percentage score

It processes the input file and writes results to the output file; see the contents of the `student.out`. The input file `student.in` contains the following information:

```
6
Rajkumar
84
Tejaswi
82
Smrithi
60
Anand
55
Rajshree
40
Ramesh
33
```

The above *Run* has created the output file `student.out` containing the following:

```
          Students Information Processing
------------------------------------------
Name: Rajkumar
Percentage: 84
Passed in: First class with distinction
------------------------------------------
Name: Tejaswi
Percentage: 82
Passed in: First class with distinction
------------------------------------------
Name: Smrithi
Percentage: 60
Passed in: First class
------------------------------------------
Name: Anand
Percentage: 55
Passed in: Second class
------------------------------------------
Name: Rajshree
Percentage: 40
Passed in: Third class
------------------------------------------
Name: Ramesh
Percentage: 33
Passed in: Sorry, Failed!
------------------------------------------
```

In `main()`, the statements

```
fstream infile;  // input file
fstream outfile; // output file
```

create objects of the stream class `fstream`, and the statements

```
infile.open( "student.in", ios::in );
outfile.open( "student.out", ios::out );
```

bind the stream objects `infile` and `outfile` to disk files named `student.in` and `student.out` respectively. Note that the stream objects `infile` and `outfile` are instances of the `fstream` class, but they are opened in different modes i.e., `infile` is opened in the *read mode*, whereas `outfile` is opened in the *write mode*. The statement

```
infile >> name;
```

reads name string from the input disk file, and the statement

```
outfile << "Name: " << name << endl;
```

writes the same to the output disk file. The file processing is carried on until all the records are processed. Note that the syntax for writing to the disk file resembles that used for writing to the console.

18.11 Random Access to a File

The program `fio.cpp` handles files using the `fstream` class. It uses `fstream` to perform both input-output operation on the `test.del` file. Since, the class `fstream` is derived from `iostream`, both input and output can be done on the same stream (same file in this case).

```
// fio.cpp: Input and output operations on file, random access
#include <iostream.h>
#include <fstream.h>
#define READ_SIZE 6
void main()
{
   char reader[ READ_SIZE + 1 ];
   // fstream constructor, open in binary input and output mode
   fstream fstr( "test.del", ios::binary|ios::in|ios::out );
   // Write the numbers 0 to 9 to file
   for( int i = 0; i < 10; i++ )
      fstr << i;
   // Set the write (put) pointer.
   fstr.seekp( 2 );
   fstr << "Hello";
   // Set the read (get) pointer.
   fstr.seekg( 4 );
   fstr.read( reader, READ_SIZE );
   reader[ READ_SIZE ] = 0;      / end of string
   cout << reader << endl;
}
```

Run

```
llo789
```

Note that an instance of `fstream` has two file pointers associated with it: a `get` pointer used while reading, and a `put` file pointer used while writing. The statement

```
fstr.seekp( 2 );
```

sets the `put` pointer to an offset 2.

The program first writes ASCII codes of the digits 0 to 9 to the file `test.del`, moves the *put pointer* by an offset 2 from the beginning of the file and the overwrites the numbers 3 through 6 with the string "`Hello`". It then reads 6 characters from the offset 4 into the array `reader`. The last line of the program will display these 6 characters, which will be `llo789`. After all writes are completed, the contents of the file `test.del` will be: `01Hello789`

The facility for direct file processing is essential in database applications. They perform extensive data read, write, update, and search activities. These actions require movement of the file pointers (get or put) from one position to another. This can be easily performed by using the `seek()`, `read()`, and `write()` functions.

The location at which the mth object is stored can be computed using a relation:

*location = m * sizeof(object)*

This specifies the offset at which the object is stored in a file. It can be used to manipulate the mth object by using the `read()` or `write()` functions.

The program `direct.cpp` illustrates the mechanism of updating a file by random access. It uses the file `person.txt` to store objects and then these objects can be updated if necessary. The file pointers *get* and *put* are positioned based on the object to be accessed.

```
// direct.cpp: accessing objects randomly
#include <fstream.h>
#include <ctype.h>       // For toupper
#include <string.h>      // For strlen
#define MAXNAME 40
class Person
{
  private:
     char name[ MAXNAME ];
     int age;
  public:
     // this function writes the class's data members to the file
     void write( ofstream &os )
     {
        os.write( name, strlen( name ) );
        os << ends;
        os.write( (char*)&age, sizeof( age ) );
     }
     // this function reads the class's date member from the file.
     // It returns nonzero if no errors were encountered while reading.
     int read( ifstream &is )
     {
        is.get( name, MAXNAME, 0 );
        name[ is.gcount() ] = 0;
        is.ignore( 1 );   // ignore the NULL terminator in the file.
        is.read( (char*)&age, sizeof( age ) );
        return is.good();
     }
     // stream operator, << overloading
     friend ostream & operator << ( ostream &os, Person &b );
     // stream operator >> operator overloading
     friend istream &operator >> ( istream &is, Person &b );
     // output file stream operator overloading
};
istream &operator >> ( istream &is, Person &b )
{
   cout << "Name: ";
   is >> ws;   // flush input buffer
   is.get( b.name, MAXNAME );
   cout << "Age : ";
   is >> ws >> b.age;
   return is;
}
ostream &operator << ( ostream &os, Person &b )
{
   os << "Name: " << b.name << endl;
   os << "Age : " << b.age << endl;
   return os;
}
```

```
void main()
{
   Person p_obj;
   int count, obj_id;
   cout << "Database Creation..." << endl;
   // open a file in binary mode and write objects to it
   ofstream ofile( "person.dat", ios::trunc|ios::binary );
   count = 0;
   char ch;
   do
   {
      cout << "Enter Object " << count << " details..." << endl;
      cin >> p_obj;
      count = count + 1;
      // write object to the output file
      ofile.write( (char *) &p_obj, sizeof( p_obj ));
      cout << "Another ? ";
      cin >> ch;
   } while( toupper( ch ) == 'Y' );
   ofile.close();
   // Output loop, display file content
   fstream iofile( "person.dat", ios::binary|ios::in|ios::out );
   cout << "Database Access..." << endl;
   while( 1 )
   {
      cout << "Enter the object number to be accessed <-1 to end>: ";
      cin >> obj_id;
      if( obj_id < 0 || obj_id >= count )
         break;
      int location = obj_id * sizeof( p_obj );
      iofile.seekg( location, ios::beg );
      iofile.read( (char *) &p_obj, sizeof( p_obj ) );
      cout << p_obj;
      cout << "Wants to Modify ? ";
      cin >> ch;
      if( ch == 'y' || ch == 'Y' )
      {
         cin >> p_obj;
         // update the object in the file
         iofile.seekp( location, ios::beg );
         iofile.write( (char *) &p_obj, sizeof( p_obj ) );
      }
   }
   iofile.close();
}
```

Run

```
Database Creation...
Enter Object 0 details...
Name: Rajkumar
Age : 25
```

```
Another ? y
Enter Object 1 details...
Name: Tejaswi
Age : 20
Another ? y
Enter Object 2 details...
Name: Kalpana
Age : 15
Another ? n
Database Access...
Enter the object number to be accessed <-1 to end>: 0
Name: Rajkumar
Age : 25
Wants to Modify ? n
Enter the object number to be accessed <-1 to end>: 1
Name: Tejaswi
Age : 20
Wants to Modify ? y
Name: Tejaswi
Age : 5
Enter the object number to be accessed <-1 to end>: 1
Name: Tejaswi
Age : 5
Wants to Modify ? n
Enter the object number to be accessed <-1 to end>: -1
```

In the program, initially a database is created without supporting its modification during creation. After creating the database file, the object `iofile` of class `fstream` is created using the statement,

```
fstream iofile( "person.dat", ios::binary|ios::in|ios::out );
```

It connects the file `person.dat` to the stream based object and permits both the read and write operations to be performed on the same file.

To read objects randomly, there must be a mechanism for converting object-id (object request) into the location at which it is stored. This is achieved by computing the location of the object storage using the relation :

```
int location = obj_id * sizeof( p_obj );
```

and *put pointer* is set to this by:

```
iofile.seekg( location, ios::beg );
```

and the statement:

```
iofile.read( (char *) &p_obj, sizeof( p_obj ) );
```

reads the file and stores into the object.

18.12 In-Memory Buffers and Data Formatting

The C's I/O system has two functions: `sscanf()` and `sprintf()` (whose prototypes appear in the `stdio.h` header file) for formatted I/O with memory buffers. The function `sscanf` performs formatted input from a character array, and `sprintf` does formatted output to a character array. These functions are normally used while displaying numbers in graphical environments (like BGI and Windows) where the output functions accept only strings.

C++ supports stream classes (declared in strstrea.h): istrstream (handling input of data from the array), ostrstream (handling output of data to the array), and strstream (transfer of data both ways) to handle character arrays in memory. In many cases, these streams may be easier to use than ordinary strings, since their buffers are dynamic. These streams can be used with stream operators, manipulators, etc., in the same way as the file streams. But their constructors have different specification. The program cmdadd.cpp illustrates the use of istrstream class in creating stream buffers and using it for extracting the data. It adds all the numbers passed as command line arguments.

```
// cmdadd.cpp: addition of numbers passed through command line
#include <strstrea.h>
void main( int argc, char *argv[] )
{
  int i = 1;
  long num, sum=0;
   if( argc < 2 )
   {
     cout << "Usage: cmdadd list_of_numbers_to_be_added";
     return;
   }
   while( --argc )
   {
     istrstream arg( argv[ i ] );
     arg >> num;
     sum += num;
     i++;
  }
  cout << sum << endl;
}
```

Run

At System prompt: cmdadd 1 2 3
6

In main(), the statement

    istrstream arg(argv[i]);

creates an object of the class istrstream and connects the same to a buffer. This object can now be used to read data from the associated buffer. The statement

    arg >> num;

extracts the data value and stores into the variable num. This method of accessing data is similar to performing I/O with the console and a file.

18.13 Error Handling During File Manipulations

In the real time environment, many users access different files without any predefined access pattern. The following are the different situations that can arise while manipulating a file:

- Attempting to open a non-existent file in read-mode.
- Trying to open a read-only marked file in write-mode.
- Trying to open a file with invalid name.

- Attempting to read beyond the end-of-the-file.
- Sufficient disk space is not available while writing to a file.
- Attempting to manipulate an unopened file.
- Stream object created but not connected to a file.
- Media (disk) errors reading/writing a file.

Such conditions must be detected while manipulating files and appropriate action should be taken to achieve consistent access to files.

Every stream (`ifstream`, `ofstream`, and `fstream`) has a *state* associated with it. Errors and nonstandard conditions are handled by setting and testing this state appropriately. The stream status variable and information recorded by its bits is shown in Figure 18.11.

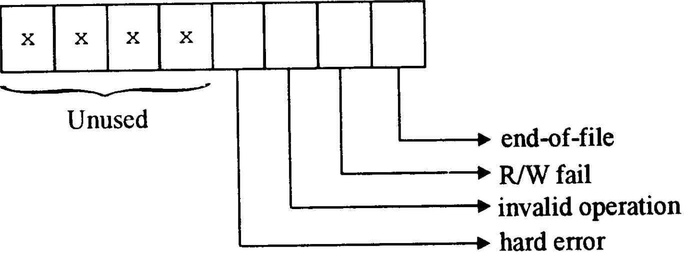

Figure 18.11: State variable format

The `ios` class supports several functions to access the status recorded in the data member `io_state`. These functions and the meaning of their return values are shown in Table 18.5.

Function	Meaning of Return Value
`eof()`	TRUE, (non-zero) if EOF encountered while reading FALSE, (zero) otherwise
`fail()`	TRUE, if read or write operation has failed; FALSE, otherwise
`bad()`	TRUE, invalid operation is attempted or any unrecoverable errors FALSE, otherwise however, it can be recovered
`good()`	TRUE, if operation is successful i.e., all the above are functions that return false., if `file.good()` is true, everything is fine and can proceed for further processing
`rdstate()`	returns the status-state data member of the class `ios`
`clear()`	clear error states and further operations can be attempted

Table 18.5: Error handling functions and their return values

The following examples illustrates the mechanism for checking errors during file operations:

1. Opening a non-existent file in read mode:

```
ifstream infile( "myfile.dat" );
if( !infile )
```

```
{
   // file does not exist
}
```

2. Open fail: opening read-only marked file

```
ofstream outfile( "myfile.dat" );
if( !infile )        // or if( infile.bad() )
{
   // file already exist and marked as read only
}
```

3. Detecting end of file

```
while( !infile.eof() )   // processes until end-of-file is reached
{
   // process file
}
```

4. Read fail

```
infile.read(...);
if( infile.bad() )
{
   // file cannot be processed further
}
```

5. Invalid filename

```
infile.open( "|-*" );
if( !infile )
{
   // invalid file name
}
```

6. Processing unopened file

```
infile.read(..);    // read file
if( infile.fail() )
{
   // file is not opened
}
```

The program `outfile.cpp` illustrates the trapping of all possible errors, which may be encountered during file processing.

```
// outfile.cpp: writes all the input into the file 'sample.out'
#include <fstream.h>
#include <process.h>
#include <string.h>
void main()
{
   char buff[ 80 ];
   ofstream outfile; // output file
   outfile.open("sample.out");   // open in output mode
   if( outfile.bad() )  // open fail
   {
     cout << "Error: sample.out unable to open";
     exit( 1 );
```

```
    }
    // loop until input = "end"
    while(1)
    {
       cin.getline(buff, 80);  // read a line from keyboard
       if( strcmp( buff, "end" ) == 0 )
          break;
       outfile << buff << endl;   // write to output file
       if( outfile.fail() )
       {
          cout << "write operation fail";
          exit( 1 );
       }
    }
    outfile.close();
}
```

Run

```
OOP is good
C++ is OOP
C++ is good
end
```

Note: On execution of the above program, the file `sample.out` contains the following information entered through the standard input device, keyboard:

```
OOP is good
C++ is OOP
C++ is good
```

In `main()`, the statement

```
ofstream outfile;  // output file
```

creates the object `outfile` and the statement

```
outfile.open("sample.out");   // open in output mode
```

opens the file `sample.out` in the output mode. The statement

```
if( outfile.bad() )    // open fail
```

checks for the status of the file open command. If open fails, it returns `1`, otherwise `0`. The statement

```
outfile << buff << endl;   // write to output file
```

writes the contents of the variable `buff` followed by a new-line character to the file. The statement

```
if( outfile.fail() )
```

checks for the status of the preceding write operation.

18.14 Filter Utilities

The operating system provides many tools for browsing through the contents of the file, copying one file to another, printing files on the printer, and beautifying the content of files. Such utilities are called filter utilities because of their nature of filtering input files and presenting them in an appealing form. For instance, the `more` command (DOS or UNIX) display the contents of the files page by page on the

console. Using the services of C++ streams such filter utilities can be built. Filter utilities are designed usually to accept the name of a file to be processed through the *command-line arguments.*

The command-line arguments are entered by the user at the shell prompt, and are delimited by white-space. (The first argument is a name of the command; filename containing the executable program.) These arguments are passed to the `main()` function of the program with the following syntax:

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

The first argument `argc` represents the argument count, whereas, the second argument is a pointer to an argument vector. For instance, when the following command is issued at the shell prompt,

```
copy boy.exe girl.exe
```

the value of `argc` and `argv` are as follows:

```
argc = 3
argv[0] = copy
argv[1] = boy.exe
argv[2] = girl.exe
```

The program `cp.cpp` is designed as a filter utility. It copies the source file into another destination file in the disk. The names of the source and destination files have to passed through the command line arguments. It can be used to copy both the ASCII and BINARY files.

```
// cp.cpp: Copy a file to another file
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
#include <process.h>
const int BUFFSIZE = 512;
int CopyFile( char *SourceFile, char *DestinationFile )
{
   fstream infile;  // source file
   fstream outfile; // destination file
   char buff[ BUFFSIZE + 1 ];
   // Open source file for reading
   infile.open( SourceFile, ios::in | ios::binary );
   if( infile.fail() )
   {
      cout << "Error: " << SourceFile << " non-existent";
      return 1; // no input file
   }
   outfile.open( DestinationFile, ios::out | ios::binary );
   if( outfile.fail() )
   {
      cout << "Error: " << DestinationFile << " unable to open";
      return 2; // cannot be written to a destination file
   }
   while( !infile.eof() )
   {
      infile.read( (char *) buff,  BUFFSIZE );
      outfile.write( (char *) buff, infile.gcount() );
      if( infile.gcount() < BUFFSIZE )
         break;
   }
   infile.close();
```