Files - **Introduction**

A file is collection of data or information that has a name, called the filename. Files are stored in secondary storage devices such as floppy disks and hard disks.

The main memories of a computer such as random access memory or read-only memory are not used for the storage of files. This is because the main memory of a computer is limited and cannot hold a large amount of data. Another reason is that the main memory is volatile; that is, when the computer is switched off, the contents of RAM vanish.

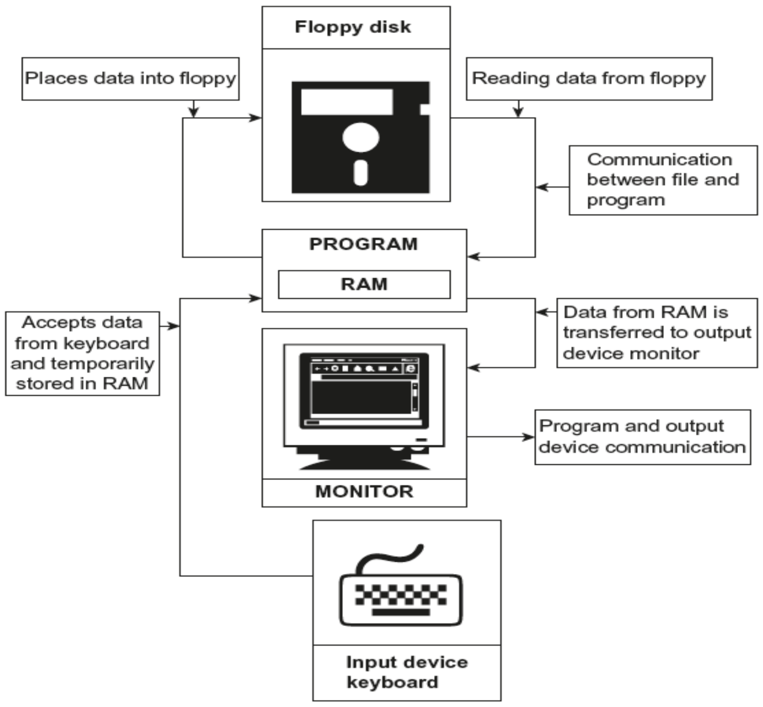

Floppy disk
Places data into floppy
Reading data from floppy
Communication between file and program
PROGRAM
RAM
Accepts data from keyboard and temporarily stored in RAM
Data from RAM is transferred to output device monitor
Program and output device communication
MONITOR
Input device keyboard

Fig. Communication between program, file, and output device

As shown in Figure, the data read from the keyboard are stored in variables. Variables are created in RAM (type of primary memory).. It is also possible to read data from secondary storage devices. When data are read from such devices, they are placed in the RAM and then, console I/O operations are used to transfer them to the screen. RAM is used to hold data temporarily.

Data communication can be performed between programs and output devices or between files and programs. File streams are used to carry the communication among the above-mentioned devices. The stream is nothing but a flow of data in bytes in sequence. If data were received from input devices in sequence, then it is called a ***source stream***, and if the data were passed to output devices, then it is called a ***destination stream***. Figure shows the input and output streams. The input stream brings data to the program, and the output stream collects data from the program. In another way, the input stream extracts data from the file and transfers it to the program; whereas the output stream stores the data in the file provided by the program.

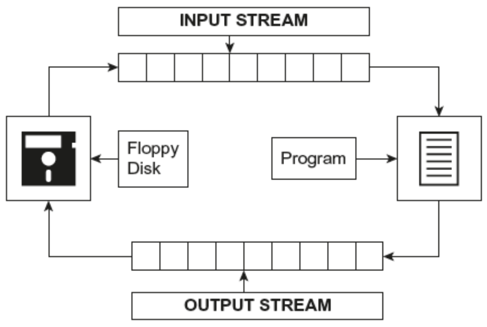
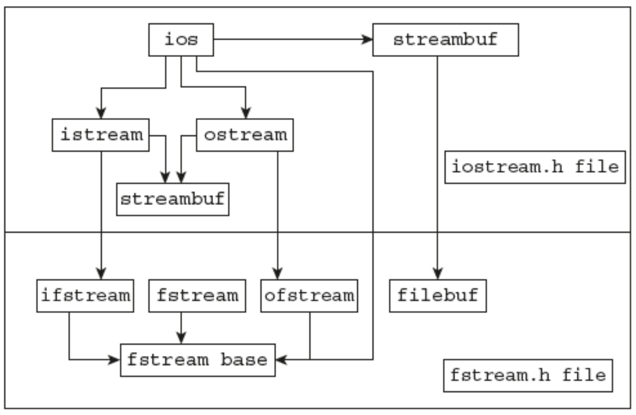


Fig. Input and output streams

File Stream Classes

A stream is nothing but a flow of data. In the object-oriented programming, the streams are controlled using the classes.



The **ios** class is the base class. All other classes are derived from the **ios** class. These classes contain several member functions that perform input and output operations. The **streambuf** class has low-level routines and provides interface to physical devices.

The **istream** and **ostream** classes control input and output functions, respectively. The ios is the base class of these two classes. The functions get(), getline(), and read() and overloaded extraction operators (>>) are defined in the **istream** class. The functions put(), write()**,** and overloaded insertion operators (<<) are defined in the **ostream** class. The **iostream** class is also a derived class. It is derived from istream and ostream classes. The classes ifstream, ofstream and fstream are derived from **istream** ,**ostream** and **iostream** respectively. These classes handle input and output with the disk files. The header file fstream.h contains a declaration of **ifstream, ofstream,** and **fstream** classes, including isotream.h file. This file should be included in the program while doing disk I/O operations.

Details of File Stream Classes:

Class	**Description**
filebuf	Sets the file buffers to read and write. It holds constant openprot used in function open() and close() as a member.
fstreambase	The fstreambase acts as a base class for fstream, ifstream, and ofstream. The functions such as open() and close() are defined in fstreambase
ifstream	Provides input operations on files. Contains open() with default input mode. Inherits the functions as get(), getline(), seekg(), tellg()**,** and read() from istream class
ofstream	Provides output operations on files. Contains open() with default output mode. Inherits the functions as put(), seekp(), write(), and tellp()from ostream class
fstream	Provides support for simultaneous input/output file stream class. Inherits all functions from istream and ostream classes through iostream.

Steps of File Operations

Before performing file operations, it is necessary to create a file. The operation of a file involves the following basic activities:

- Specifying suitable file name
- Opening the file in desired mode
- Reading or writing the file (file processing)
- Detecting errors
- Closing the file

File Opening: In order to perform operations, we have to create a file stream object and connecting it with the file name. The classes **ifstream, ofstream,** and **fstream** can be used for creating a file stream. The selection of the class is according to the operation that is to be carried out with the file. The operation may be read or write. Two methods are used for the opening of a file. They are as follows:

- Constructor of the class
- Member function open()

1. Constructor of the class:

When objects are created, a constructor is automatically executed, and objects are initialized. In the same way, the file stream object is created using a suitable class, and it is initialized with the file name. The constructor itself uses the file name as the fist argument and opens the file. The class ofstream creates output stream objects, and the class ifstream creates input stream objects.

Consider the following examples:

a) ofstream out (“text”);
b) ifstream in(“list”);

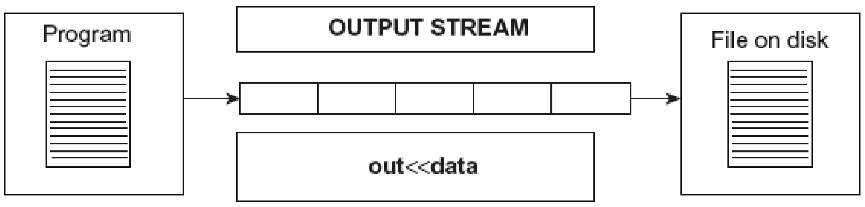
In the statement (a), out is an object of the class ofstream; file name text is opened, and data can be written to this file. The file name text is connected with the object out. Similarly, in the statement (b), in is an object of the class ifstream. The file list is opened for input and connected with the object in. It is possible to use these file objects in program statements such as stream objects. Consider the following statements:

```
cout<<"One Two Three";
```

The above statement displays the given string on the screen.

```
out<<"One Two Three";
```

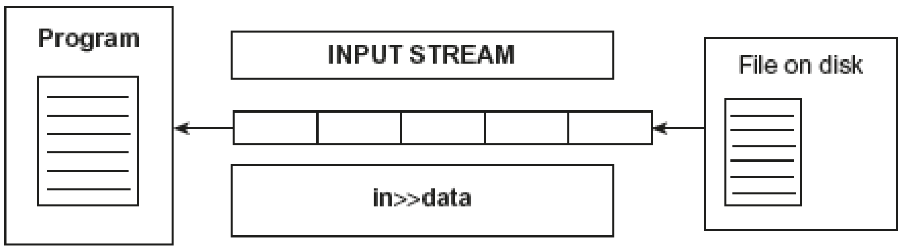
The above statement writes the specified string into the file pointed by the object out as shown in Figure. The insertion operator << has been overloaded appropriately in the ostream class to write data to the appropriate stream.



Similarly, in the following statements,

```
in>>string;   // Reads data from the file into string where string is a character array
in>>num;      // Reads data from the file into num where num is an integer variable
```

the in object reads data from the file associated with it, as shown in figure. For reading data from a file, we have to create an object of the **ifstream** class.



/* Write a program to open an output file using fstream class.*/

```
#include <iostream>
#include <fstream>
using namespace std;
int  main()
{
          char name[15];
         int age;
         ofstream out("text");
         cout<<"Enter Name:"<<endl;
         cin>>name;
         cout<<"Enter Age:"<<endl;
         cin>>age;
         out<<name<<"\t";
         out<<age <<endl;
         out.close(); // File is closed
          return 0;
}
```

Explanation*:* In the above program, the statement `ofstream out (“text”)` text is opened and connected with the object `out.`

Contents of the file text: pvpsit 15

/* Write a program to read data from file using object of ifstream class.*/

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

int main() {
         string name;
         int age;
         ifstream in("text"); // Opens a file in read mode
         in>>name;
         in>>age;
         cout<<"Name:"<<name<<endl;
         cout<<"Age:"<<age;
         in.close();
         return 0;
}
```

Output:

```
         pvpit
         15
```

/* Write a program to write and read data from file using object of fstream class.*/

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

int main() {
        string name;
        int age;
        fstream f("text");
        cout<<"Enter Name:"<<endl;
        cin>>name;
        cout<<"Enter Age:"<<endl;
        cin>>age;
        f<<name<<"\t";
        f<<age <<endl;

        f>>name;
        f>>age;
        cout<<"\nName:"<<name<<endl;
        cout<<"Age:"<<age;
        f.close();
        return 0;
}
```

In the above programs, the file associated with the object are automatically closed when the stream object goes out of scope. In order to explicitly close the file, the following statement is used:

```
out.close();
in.close();
```

Here, out is an object, and close() is a member function that closes the file connected with the object out. Similarly, the file associated with the object in is closed by the member function close().

/* **Write a program to write and read text in a file. Use ofstream and ifstream classes.*/**

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        string name;
        int age;
        ofstream out("text");
        cout<<"Enter Name:"<<endl;
        cin>>name;
        cout<<"Enter Age:"<<endl;
        cin>>age;
        out<<name<<"\t";
        out<<age <<endl;
        out.close(); // File is closed
        ifstream in ("text");
        in>>name;
        in>>age;
        cout<<"\nName:"<<name<<endl;
        cout<<"Age:"<<age;
        in.close();
        return 0;
}
```

Output:

```
Enter Name : PVPSIT
Enter Age : 24
Name : PVPSIT
Age : 24
```

2. The open() function

The open() function is used to open a file, and it uses the stream object. The open() function has two arguments. First is the file name, second is the mode and this is optional. The mode specifies the purpose of opening a file; that is, read, write, append, and so on. If we don’t specify any mode default will be considered.

In the following examples, the default mode is considered. The default values for ifstream is ios::in reading only and for fstream is ios::out writing only.

(A) Opening file for write operation

```
ofstream out;        // Creates stream object out
out.open (“marks”);        // Opens file and links with the object out
out.close()        // Closes the file pointed by the object out
out.open (“result”);        // Opens another file
```

(B) Opening file for read operation

```
ifstream in;                    // Creates stream object in
in.open (“ marks”);             // Opens file and link with the object in
in.close() ;                    // Closes the file pointed by object in
```

/* Write a program to open the file for writing and reading purposes. Use open() function.*/

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

int main() {
        string name;
        int age;
        ofstream out;
        out.open("Text");
        cout<<"Enter Name:"<<endl;
        cin>>name;
        cout<<"Enter Age:"<<endl;
        cin>>age;
        out<<name<<"\t";
        out<<age <<endl;
        out.close(); // File is closed
        ifstream in;
        in.open("Text");
        in>>name;
        in>>age;
        cout<<"\nName:"<<name<<endl;
        cout<<"Age:"<<age;
        in.close();
        return 0;
}
```

Output:

```
Enter Name: PVPSIT
Enter Age: 21
Name:PVPSIT
Age:21
```

/* **Program to create a file consisting of 'n' employee's details and print employee infromation.***/

```
#include <iostream>
#include<fstream>
#include<iomanip>
using namespace std;
class emp{
        int empno;
        string name;
        float sal;
public:
        void get();
        void display();
};
void emp::get()
{
        cout<<"Enter empno,name,salary"<<endl;
        cin>>empno>>name>>sal;
}
void emp::display()
{
        cout<<"\t"<<empno<<"\t"<<name<<"\t"<<sal<<endl;
}
int main() {
        ofstream fout;
        emp e;
        int i,n;
        fout.open("emp.txt",ios::out);
        cout<<"Enter Number of records";
        cin>>n;
        cout<<"Enter "<<n<<"employee details";
        for(i=1;i<=n;i++)
        {
                e.get();
                fout.write((char *)&e,sizeof(e));
        }
        fout.close();
        cout<<"writing finished"<<endl;
        cout<<"The data in the file is"<<endl;

        ifstream fin;
        fin.open("emp.txt",ios::in);
        while(fin)
        {

                fin.read((char *)&e,sizeof(r));
                e.display();
        }
        fin.close();
        return 0;

}
```

Checking for Errors

Various errors can be made by the user while performing a file operation. Such errors should be reported in the program to avoid further program failure. When a user attempts to read a file that does not exist or opens a read-only file for writing purpose, the operation fails in such situations. Such errors should be reported, and proper actions have to be taken before further operations are performed.

The ! (logical negation operator) overloaded operator is useful for detecting errors. It is a unary operator and, in short, it is called a `not` operator. The (!) `not operator` can be used with objects of stream classes. This operator returns a non-zero value if a stream error occurs during an operation. Consider the following program:

/*Write a program to check whether the file is successfully opened or not.*/

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

int main()
{
        ifstream in ("text");
        if (!in) cerr <<"File is not opened";
        else cerr <<"File is opened";
        return 0;
}
```

Output: File is not opened

Finding End of a File

While reading data from a file, it is necessary to find where the file ends, that is, the end of the file. The programmer cannot predict the end of the file. If in a program, while reading the file, the program does not detect the end of the file, the program drops in an infinite loop. To avoid this, it is necessary to provide correct instructions to the program that detects the end of the file. Thus, when the end of the file is detected, the process of reading data can be easily terminated. The eof() member function() is used for this purpose.

The eof() stands for the end of the file. It is an instruction given to the program by the operating system that the end of the file is reached. It checks the ios::eofbit in the ios::state. The eof() function returns the non-zero value, when the end of the file is detected; otherwise, it is zero.

/***Write a program to read and display contents of file. Use eof() function.*/**

```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
        ofstream ofs;
        char  ch;
        ofs.open("hello.txt");
        cout<<"Enter some data at end type q(QUIT)"<<endl;
        cin>>ch;
        while(ch!='q')
        {
                ofs<<ch;
                cin>>ch;
        }
        ofs.close();
        ifstream ifs;
        ifs.open("hello.txt");
        cout<<"The Data from the file"<<endl;
        while(!ifs.eof())
        {
                ifs>>ch;
                cout<<ch;
        }
        ifs.close();
        return 0;
}
```

Error Handling Functions

Errors may occur due to

1. An attempt to read a file that does not exist
2. The file name specified for opening a new file may already exist
3. An attempt to read the contents of a file when the file pointer is at the end of the file
4. Insufficient disk space
5. Invalid file name specified by the programmer
6. A file opened may be already opened by another program
7. An attempt to open the read-only file for writing operation
8. Device error

The **stream state** member from the **class ios** receives values from the status bit of the active file. The class `ios` also contains many different member functions. These functions read the status bit of the file when an error occurred during program execution.

All streams such as ofstream, ifstream, and fstream contain the state connected with them.

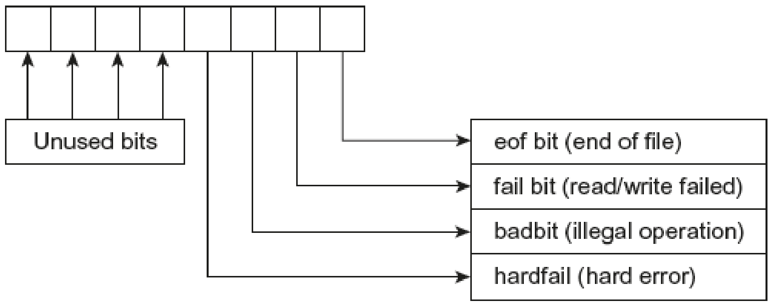


Fig. Status bits

eof bit	End of file encountered.
fail bit	Operation unsuccessful
bad bit	Illegal operation due to wrong size of buffer
hard fail	Critical error

Error Trapping Functions

Functions	Working and return value
fail()	Returns non-zero value if an operation is unsuccessful. This is carried out by reading the bits ios::fail bit, ios:: bad bit, and ios::hard bit.
eof()	Returns non-zero value when the end of the file is detected; otherwise, it returns zero. This is carried out by checking ios::eof bit.
bad()	Returns non-zero value when an error is found in the operation. The ios::bad bit is checked.
good()	Returns non-zero value if no error occurred during the file operation, that is, no status bits were set.
rdstate()	Read the stream state and returns the values.
clear(int=0)	To clear particular bit(s), clear() clears all the bits. clear(ios::fail) clears only fail bit.

/* Write a C++ program to display status of various errors trapping functions. */

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

int main() {
        ifstream in;
        in.open("suresh1.dat");
        cout<<"File"<<in<<endl;
        cout<<"rdstate:"<<in.rdstate()<<endl;
        cout<<"fail():"<<in.fail()<<endl;
        cout<<"eof():"<<in.fail()<<endl;
        cout<<"bad():"<<in.bad()<<endl;
        cout<<"good():"<<in.good()<<endl;
        in.close();
        return 0;
}
```

Output:

```
File:0
rdstate:4
fail():1
eof():1
bad():0
good():0
```

Explanation: In the above program, an attempt is made to open a non-existent file. The if statement checks the value of the object in. The specified file does not exist; hence, it displays the message 0 (File not found). The program also displays the values of various bits using the functions good(), eof(), bad(), fail(), and rdstate() error trapping functions.

FILE OPENING MODES

In previous examples, we have learned how to open files using constructors and the `open()` function using the objects of `ifstream` and `ofstream` classes. The opening of the file also involves several modes depending on the operation to be carried out with the file. The `open()` function has the following two arguments:

```
Syntax of open() function

object.open ( “file_ name”, mode);
```

Here, the object is a stream object, followed by the `open()` function. The bracket of the open function contains two parameters. The first parameter is the name of the file, and the second is the mode in which the file is to be opened. In the absence of a mode parameter, a default parameter is considered. The file mode parameters are as shown in Table 16.1.

Table 16.1 File modes

Mode parameter	**Operation**
ios::app	Adds data at the end of file
ios::ate	After opening character pointer goes to the end of file
ios:: binary	Binary file
ios::in	Opens file for reading operation
ios::nocreate	Opens unsuccessfully if the file does not exist
ios::noreplace	Opens files if they are already present
ios::out	Open files for writing operation
ios::trunc	Erases the file contents if the file is present

1. The mode `ios::out` and `ios::trunc` are the same. When `ios::out` is used, if the specified file is present, its contents will be deleted (truncated). The file is treated as a new file.
2. When the file is opened using `ios:app` and `ios::ate` modes, the character pointer is set to the end of the file. The `ios:: app` lets the user add data at the end of the file, whereas the `ios:ate` allows the user to add or update data anywhere in the file. If the given file does not exist, a new file is created. The mode `ios::app` is applicable to the output file only.

3. The ifstream creates an input stream and an ofstream output stream. Hence, it is not compulsory to give mode parameters.
4. While creating an object of the `fstream` class, the programmer should provide the mode parameter. The fstream class does not have the default mode.
5. The file can be opened with one or more mode parameters. When more than one parameter is necessary, a bit-wise OR operator separates them. The following statement opens a file for appending. It does not create a new file if the specified file is not present.

```
File opening with multiple attributes

out.open (“file1”, ios::app | ios:: nocreate)
```

16.9 Write a program to open a file for writing and store `float` numbers in it.

```
#include<fstream.h>

#include<iomanip.h>


void main()

{

float a=784.52, b=99.45,c =12.125;

ofstream out (“float.txt”,ios::trunc);

out<<setw(10)<<a<<endl;

out<<setw(10)<<b<<endl;

out<<setw(10)<<c<<endl;

}
```

Explanation: In the above program, the file `“float.txt”` is opened. If the file already exists, its contents are truncated. The three `float` numbers are written in the file.

16.10 Write a program to open a file in binary mode. Write and read the data.

```
#include<fstream.h>

#include<conio.h>


int main()

{

clrscr();

ofstream out;

char data[32];

out.open (“text”,ios::out | ios::binary);

cout<<“\n Enter text”<<endl;

cin.getline(data,32);

out <<data;

out.close();

ifstream in;

in.open(“text”, ios::in | ios::binary);

cout<<endl<<“Contents of the file \n”;

char ch;

{

ch= in.get();

cout<<ch;

}
return 0;

}
```

OUTPUT

Programming In ANSI and TURBO-C

Contents of the file

Programming In ANSI and TURBO-C

Explanation: The above program is similar to the previous one. The only difference is that here files are opened in binary mode.

16.7 FILE POINTERS AND MANIPULATORS

All file objects hold two file pointers that are associated with the file. These two file pointers provide two integer values. These integer values indicate the exact position of the file pointers in the number of bytes in the file. The read or write operations are carried out at the location pointed by these file pointers .One of them is called `get pointer (input pointer)`, and the second one is called `put pointer (output pointer)`. During reading and writing operations with files, these file pointers are shifted from one location to another in the file. The `(input) get pointer` helps in reading the file from the given location, and the output pointer helps in writing data in the file at the specified location. When read and write operations are carried out, the respective pointer is moved.

While a file is opened for the reading or writing operation, the respective file pointer input or output is by default set at the beginning of the file. This makes it possible to perform the reading or writing operation from the beginning of the file. The programmer need not explicitly set the file pointers at the beginning of files. To explicitly set the file pointer at the specified position, the file stream classes provides the following functions:

Read mode: When a file is opened in read mode, the get pointer is set at the beginning of the file, as shown in Figure 16.7. Hence, it is possible to read the file from the first character of the file.

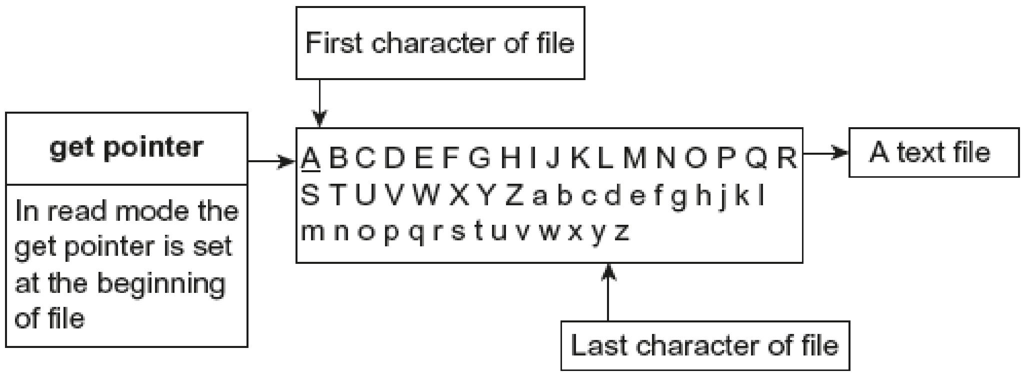

First character of file
get pointer
In read mode the get pointer is set at the beginning of file
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h j k l m n o p q r s t u v w x y z
A text file
Last character of file

Fig. 16.7 Status of get pointer in read mode

Write Mode: When a file is opened in write mode, the put pointer is set at the beginning of the file, as shown in Figure 16.8. Thus, it allows the write operation from the beginning of the file. In case the specified file already exists, its contents will be deleted.

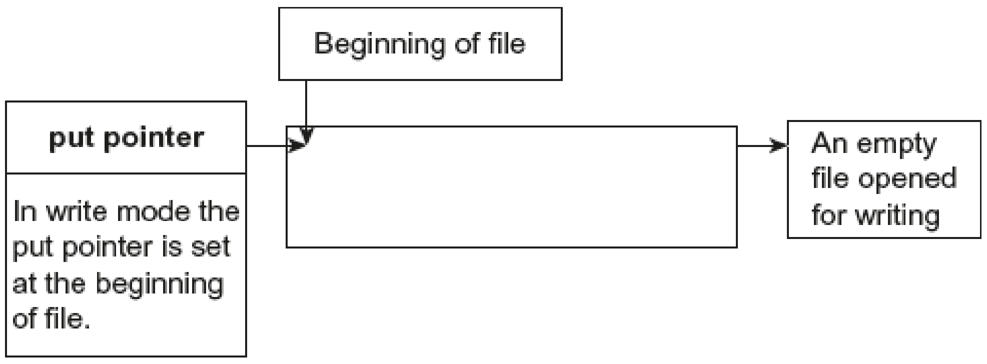

Beginning of file
put pointer
In write mode the put pointer is set at the beginning of file.
An empty file opened for writing

Fig. 16.8 Status of put pointer in write mode

Append Mode: This mode allows the addition of data at the end of the file. When the file is opened in append mode, the output pointer is set at the end of the file, as shown in Figure 16.9. Hence, it is possible to write data at the end of the file. In case the specified file already exists, a new file is created, and the output is set at the beginning of the file. When a pre-existing file is successfully opened in append mode, its contents remain safe and new data are appended at the end of the file.

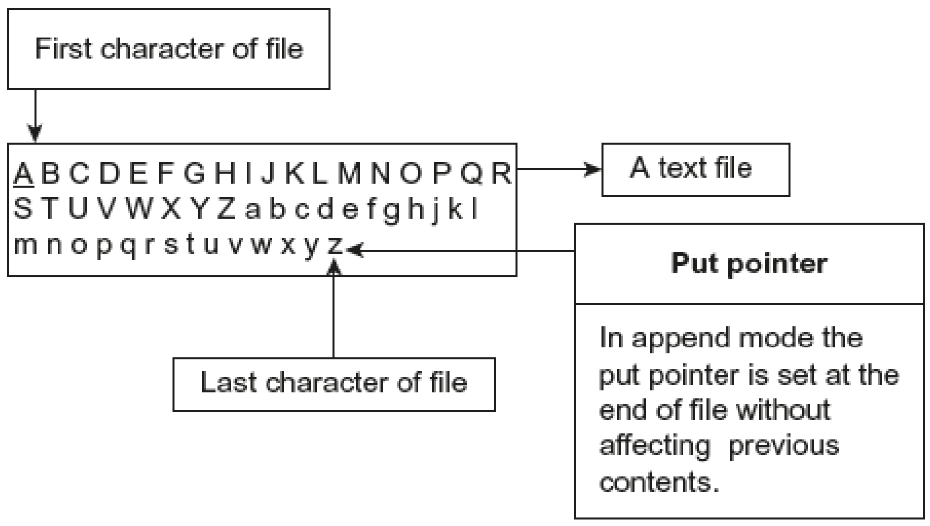

First character of file
A B C D E F G H I J K L M N O P Q R
S T U V W X Y Z a b c d e f g h j k l
m n o p q r s t u v w x y z
A text file
Put pointer
In append mode the put pointer is set at the end of file without affecting previous contents.
Last character of file

Fig. 16.9 Status of put pointer in append mode

C++ has four functions for the setting of points during file operation. The position of the curser in the file can be changed using these functions. These functions are described in Table 16.2.

Table 16.2 File pointer handling functions

Function	**Uses**	**Remark**
seekg()	Shifts input (get) pointer to a given location.	Member of ifstream class
seekp()	Shifts output (put) pointer to a given location.	Member of ofstream class
tellg()	Provides the present position of the input pointer.	Member of ifstream class
tellp()	Provides the present position of the output pointer.	Member of ofstream class

As given in Table 16.2, the `seekg()` and `tellg()` are member functions of the `ifstream` class. All the above four functions are present in the class `fstream.` The class `fstream` is derived from `ifstream` and `ofstream` classes. Hence, this class supports both input and output modes, as shown in Figure 16.10. The `seekp()` and `tellp()` work with the put pointer, and `tellg()` and `seekg()` work with the get pointer.

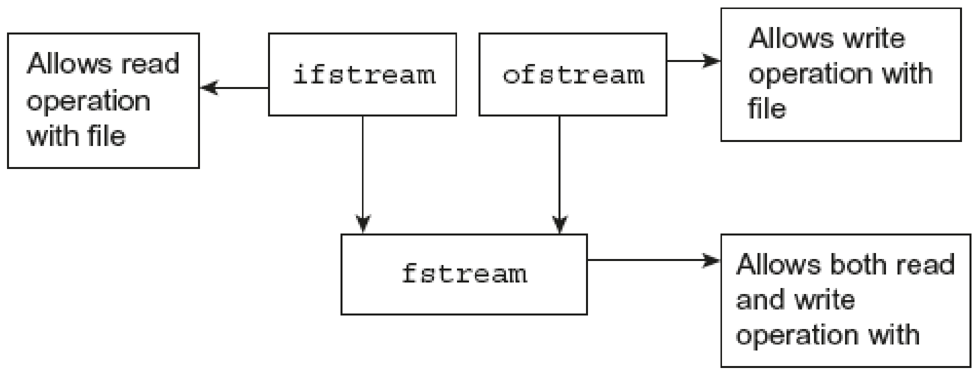

Allows read operation with file
ifstream
ofstream
Allows write operation with file
fstream
Allows both read and write operation with

Fig. 16.10 Derivation of fstream class

Now consider the following examples:

16.11 Write a program to append a file.

```
#include<fstream.h>

#include<conio.h>


int main()

{

clrscr();

ofstream out;

char data[25];

out.open (“text”,ios::out);

cout<<“\n Enter text”<<endl;

cin.getline(data,25);

out <<data;
out.close();

out.open (“text”, ios::app );

cout<<“\n Again Enter text”<<endl;

cin.getline (data,25);

out<<data;

out.close();

ifstream in;

in.open(“text”, ios::in);

cout<<endl<<“Contents of the file \n”;

while (in.eof()==0)

{

in>>data;

cout<<data;

}

return 0;

}
```

OUTPUT

```
Enter text

C-PLUS-

Again Enter text

PLUS
```

Contents of the file

C-PLUS-PLUS

Explanation: In the above program the file `text` is opened for writing, that is, output. The text read through the keyboard is written in the file. The `close()` function closes the file. Once more, the same file is opened in the append mode, and data entered through the keyboard are appended at the end of the file, that is, after the previous text. The append mode allows the programmer to write data at the end of the file. The `close()` function closes the file. The same file is opened using the object of the `ifstream` class for reading purpose. The `while` loop is executed until the end of the file is detected. The statements within the `while` loop read text from the file and display it on the screen.

16.12 Write a program to read contents of the file. Display the position of the get pointer.

```
#include<fstream.h>

#include<conio.h>


int main()

{

clrscr();

ofstream out;

char data[32];

out.open (“text”,ios::out);

cout<<“\n Enter text”<<endl;

cin.getline(data,32);

out <<data;

out.close();
ifstream in;

in.open(“text”, ios::in);

cout<<endl<<“Contents of the file \n”;

int r;

while (in.eof()==0)

{

in>>data;

cout<<data;

r=in.tellg();

cout<<“ (“<<r <<“)”;

}

return 0;

}
```

OUTPUT

```
Enter text

Programming In ANSI and TURBO-C

Contents of the file

Programming (11)In (14)ANSI (19)and (23)TURBO-C (31)
```

Explanation: The above program is similar to the previous one. In addition here, the function `tellg()` is used. This function returns the current file pointer position in the number of bytes from the beginning of the file. The number shown in brackets in the output specifies the position of the file pointer from the beginning of the file. The same program is illustrated below using the binary mode.

16.8 MANIPULATORS WITH ARGUMENTS

The `seekp()` and `seekg()` functions can be used with two arguments. Their formats with two arguments are as follows:

```
seekg(offset, pre_position);

seekp(offset, pre_position);
```

The first argument `offset` specifies the number of bytes the file pointer is to be shifted from the argument `pre_position` of the pointer. The `offset` should be a positive or negative number. The positive number moves the pointer in the forward direction, whereas the negative number moves the pointer in the backward direction. Fig 16.11 provides the status of pre-position arguments. The `pre_position` argument may have one of the following values:

- `ios::beg Beginning of the file`
- `ios::cur Current position of the file pointer`
- `ios::end End of the file`

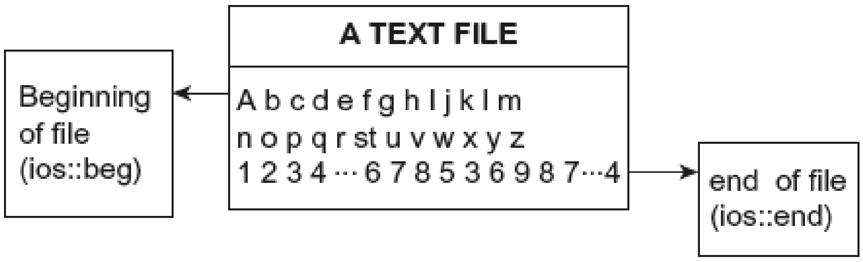


Fig. 16.11 Status of pre-position arguments

In the above figure, the status of `ios::beg` and `ios::end` is shown. The status of `ios::cur` cannot be shown to be similar to `ios::beg` or `ios::end`. The `ios::cur` means the present position of the file pointer. The `ios::beg` and `ios::end` may be referred to as `ios::cur`. Suppose the file pointer is in the middle of the file and you want to read the file from the beginning, you can set the file pointer at the beginning using `ios::beg.` However, if you want to read the file from the current position, you can use the option `ios::cur`.

The `seekg()` function shifts the associated file's input (get) file pointer. The `seekp()` function shifts the associated file's output (put) file pointer. Table 16.3 describes a few pointer offsets along with their working.

Table 16.3 File pointer with its arguments

Seek option	Working
in.seekg (0,ios :: beg)	Go to the beginning of file
in.seekg (0,ios :: cur)	Rest at the current position
in.seekg (0,ios ::end)	Go to the end of file
in.seekg (n,ios :: beg)	Shift file pointer to n+1 byte in the file
in.seekg (n,ios :: cur)	Go front by n byte from the current position
in.seekg (-n,ios :: cur)	Go back by n bytes from the present position.
in.seekg (-n,ios::end);	Go back by n bytes from the end of file

In Table 16.3, `in` is an object of the `ifstream class`.

16.13 Write a program to write text in the file. Read the text from the file from end of file. Display the contents of file in reverse order.

```
#include<fstream.h>

#include<conio.h>


int main()

{

clrscr();

ofstream out;

char data[25];

out.open (“text”,ios::out);

cout<<“\n Enter text”<<endl;

cin.getline(data,25);
out <<data;

out.close();

ifstream in;

in.open("text", ios::in);

cout<<endl<<"Reverse Contents of the file \n";

in.seekg(0,ios::end);

int m=in.tellg();

char ch;

for (int i=1;i<=m;i++)

{

in.seekg(-i,ios::end);

in>>ch;

cout<<ch;

}

return 0;

}
```

OUTPUT

```
Enter text

Visual_C_+_+

Reverse Contents of the file

+_+_C_lausiV
```

Explanation: In the above program, file `text` is opened in the output mode, and the string entered is written to the file. Again, the same file is opened for reading purpose. The statement `in.seekg (0,ios::end)`; moves the get pointer at the end of the file. The `tellg()` function returns the current position of the file pointer in the file. Hence, the file pointer is set to the end of the file. The `tellg()` returns the number of last bytes, that is, the size of the file in bytes, and it is stored in the integer variable `m`. The `for` loop executes from 1 to m. The statement `in.seekg (-i, ios::end)` reads the `ith` byte from the end of the file. The statement `in>>ch` reads the character from the file indicated by the file pointer. The `cout` statement displays the read character on the screen. Thus, the contents of the file are displayed in reverse order.

16.14 Write a program to enter a text and again enter a text and replace the first word of the first text with the second text. Display the contents of the file.

```
#include<fstream.h>

#include<conio.h>


int main()

{

clrscr();

ofstream out;

char data[25];

out.open (“text”,ios::out);

cout<<“\n Enter text”<<endl;

cin.getline(data,25);

out <<data;

out.seekp(0,ios::beg);

cout<<“\nEnter text to replace the first word of first text:”;
cin.getline(data,25);

out<<data;

out.close();

ifstream in;

in.open(“text”, ios::in);

cout<<endl<<“Contents of the file \n”;

while (in.eof()!=1)

{ in>>data;

cout<<data;

}

return 0;

}
```

OUTPUT

```
Enter text

Visual C++

Enter text to replace the first word of first text : Turbo-Contents of the file

Turbo-C++
```

Explanation: In the above program, the `text` is entered and written in the file `text.` This process is explained in the previous examples. Here again, the statement `out.seekp(0,ios::beg);` sets the file pointer (put pointer) at the beginning of the file. Again, `text` is entered and written at the current file pointer position. The previous text is overwritten.

16.9 SEQUENTIAL ACCESS FILES

C++ allows the file manipulation command to access the file sequentially or randomly. The data of the sequential file should be accessed sequentially, that is, one character at a time. In order to access the nth number of bytes, all previous characters are read and ignored. There are a number of functions to perform read and write operations with the files. Some functions read /write single characters, and some functions read/write blocks of binary data. The `put()` and `get()` functions are used to read or write a single character, whereas `write()` and `read()` are used to read or write blocks of binary data.

put() and get() functions

The function `get()` is a member function of the class `fstream.` This function reads a single character from the file pointed by the get pointer, that is, the character at the current get pointer position is caught by the `get()` function.

The function `put()` function writes a character to the specified file by the stream object. It is also a member of the `fstream` class. The `put()` function places a character in the file indicated by the put pointer.

16.15 Write a program to write and read string to the file using `put()` and `get()` functions.

```
#include<fstream.h>

#include<conio.h>

#include<string.h>


int main()

{

clrscr();

char text[50];
cout<<“\n Enter a Text:”;

cin.getline(text,50);

int l=0;

fstream io;

io.open(“data”, ios::in | ios::out);

while (l[text]!=‘\0’)

io.put(text[l++]);

io.seekg(0);

char c;

cout<<“\n Entered Text:”;

while (io)

{

io.get(c);

cout<<c;

}

return 0;

}
```

OUTPUT

```
Enter a Text : PROGRAMMING WITH C++

Entered Text : PROGRAMMING WITH C++
```

Explanation: In the above program, the file `data` is opened simultaneously in the read and write mode. The `getline()` function reads the string through the keyboard and stores it in the array `text [50]`. The statement `io.put (text[l++])` in the first `while` loop reads one character from the array and writes it to the file indicated by the stream object `io`. The first `while` loop terminates when the null character is found in the text.

The statement `io.seekg (0)` sets the file pointer at the beginning of the file. In the second `while` loop, the statement `io.get(c)` reads one character at a time from the file, and the `cout()` statement displays the same character on the screen. The `while` loop terminates when the end of the file is detected.

16.10 BINARY AND ASCII FILES

The insertion and extraction operators known as `stream operators` handle formatted data. The programmer needs to format data in order to represent them in a suitable manner. The description of formatted and unformatted data is given in Chapter 2. ASCII codes are used by the I/O devices to share or pass data to the computer system, but the central processing unit (CPU) manipulates the data using binary numbers, that is, 0 and 1. For this reason, it is essential to convert the data while accepting data from input devices and displaying the data on output devices. Consider the following statements:

```
cout<<k;   // Displays value of k on screen

cin>>k;   // Reads value for k from keyboard
```

Here, k is an integer variable. The operator << converts the value of the integer variable k into a stream of ASCII characters. In the same manner, the << operator converts the ASCII characters entered by the user into binary form. The data are entered through the keyboard, which is a standard input device. For example, you entered 21. The stream operator >> gets ASCII codes of the individual digits of the entered number 21, that is, 50 and 49. The ASCII codes of 2 and 1 are 50 and 49, respectively. The stream operator >> converts the ASCII value into its equivalent binary format and assigns it to the variable k. The stream operator << converts the value of k (21) that is stored in the binary format into its equivalent ASCII codes, that is, 50 and 49. Figure 16.12 shows a representation of integer numbers in ASCII and binary formats.

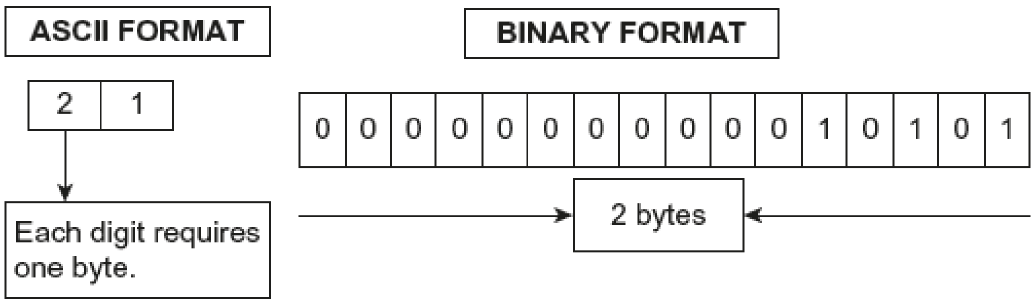


Fig. 16.12 Representation in binary and ASCII formats

16.16 Write a program to demonstrate that the data is read from the file using ASCII format.

```
#include<fstream.h>

#include<constream.h>


int main()

{

clrscr();

char c;

ifstream in(“data”);

if (!in)

{

cerr<<“ Error in opening file.”;

return 1;

}

while (in.eof()==0)

{
```

```
cout<<(char)in.get();

}

return 0;

}
```

OUTPUT

PROGRAMMING WITH ANSI AND TURBO-C

Explanation: In the above program, the data file is opened in read mode. The file already exists. Using `get()` member function of the ifstream class, the contents of the file are read and displayed. Consider the following statement:

```
cout<<(char)in.get();
```

The `get()` function reads data from the file in the ASCII format. Hence, it is necessary to convert the ASCII number into an equivalent character. The typecasting format (char) converts the ASCII number into an equivalent character. In case the conversion is not done, the output would be as follows:

```
8082797182657777737871328773847232657883733265786832848582667967-1
```

The above displayed are ASCII numbers, and –1 at the end indicates the end of the file.

After typecasting, the original string will be as shown in the output.

The write() and read() functions

The data entered by the user are represented in the ASCII format. However, the computer can understand only the machine format, that is, 0 and 1. When data are stored in the text, format numbers are stored as characters and occupy more memory space. The functions `put()` and `get()`read/ write a character. The data are stored in the file in character format. If a large amount of numeric data are stored in the file, they will occupy more space. Hence, using `put()` and `get()`creates disadvantages.

This limitation can be overcome using `write()` and `read()` functions. The `write()` and `read()`functions use the binary format of data while in operation. In the binary format, the data representation is same in both the file and the system. Figure 16.12 shows the difference between the ASCII and binary format. The bytes required to store an integer in `text` form depend on its size, whereas in the binary format the size is fixed. The binary form is accurate and allows quick read and write operations, because no conversion takes places during operations. The formats of the `write()` and `read()` function are as given below.

```
in.read((char *) & P, sizeof(P));

out.write((char *) & P, sizeof(P));
```

These functions have two parameters. The first parameter is the address of the variable `P.` The second is the size of the variable `P` in bytes. The address of the variable is converted into char type. Consider the following program:

16.17 Write a program to perform read and write operations using `write()` and `read()` functions.

```
#include<fstream.h>

#include<conio.h>

#include<string.h>


int main()

{

clrscr();

int num[]={100,105,110,120,155,250,255};
ofstream out;

out.open(“01.bin”);

out.write((char *) & num, sizeof(num));

out.close();

for (int i=0;i<7;i++) num[i]=0;

ifstream in;

in.open(“01.bin”);

in.read((char *) & num, sizeof(num));

for (i=0;i<7;i++) cout<<num[i]<<“\t”;

return 0;

}
```

OUTPUT

```
100 105 110 120 155 250 255
```

Explanation: In the above program, the integer array is initialized with `7` integer numbers. The file `“01.bin”` is opened. The statement `out.write((char *) & num, sizeof (num))` writes the integer array in the file. The `&num` argument provides the base address of the array, and the second argument provides the total size of the array. The `close()` function closes the file. Again, the same file is opened for reading purpose. Before reading the contents of the file, the array is initialized to a zero that is not necessary. The statement `in.read ((char *) & num, sizeof (num));` reads data from the file and assigns them to the integer array. The second `for` loop displays the contents of the integer array. The size of the file `“01.bin”` will be `14` bytes, that is, two bytes per integer. If the above data are stored without using the `write()` command, the size of the file will be `21` bytes.

Reading and writing class objects

The `read()` and `write()` functions perform read and write operations in a binary format that is exactly the same as an internal representation of data in the computer. Due to the capabilities of these functions, large data can be stored in a small amount of memory. Both these functions are also used to write and read class objects to and from files. During read and write operations, only data members are written to the file, and the member functions are ignored. Consider the following program:

16.18 Write a program to perform read and write operations with objects using `write()` and `read()` functions.

```
#include<fstream.h>

#include<conio.h>


class boys

{

char name [20];

int age;

float height;

public:

void get()

{

cout<< “Name:”; cin>>name;

cout<< “Age:”; cin>>age;

cout<< “Height:”; cin>>height;

}

void show()

{
```

```
cout<<“\n”<<name<<“\t”<<age <<“\t”<<height;

}

};

int main()

{

clrscr();

boys b[3];

fstream out;

out.open (“boys.doc”, ios::in | ios::out);

cout<<“\n Enter following information:\n”;

for (int i=0;i<3;i++)

{

b[i].get();

out.write ((char*) & b[i],sizeof(b[i]));

}

out.seekg(0);

cout<<“\n Entered information\n”;

cout<<“Name Age Height”;

for (i=0;i<3;i++)

{

out.read((char *) & b[i], sizeof(b[i]));

b[i].show();

}
```

```
out.close();

return 0;

}
```

OUTPUT

```
Enter following information:

Name : Kamal

Age : 24

Height : 5.4

Name : Manoj

Age : 24

Height : 5.5

Name : Rohit

Age : 21

Height : 4.5

Entered information

Name Age Height

Kamal 24 5.4

Manoj 24 5.5

Rohit 21 4.5
```

Explanation: In the above program, the `class boys` contains data members' `name, age,` and `height` of `char, int,` and `float` type. The class also contains the member functions `get()` and `show()` to read and display the data. In function `main(),` an array of three objects is declared, that is, `b [3]`. The file `“boys.doc”` is opened in the output and input mode to write and read data. The first `for` loop is used to call the member function `get()`, and data read via the `get()` function is written to the file by the `write()` function. The same method is repeated while reading the data from the file. While reading data from the file, the `read()` function is used, and the member function `show()` displays the data on the screen.

16.11 RANDOM ACCESS OPERATION

Data files always contain a large amount of information, and the information always changes. The changed information should be updated; otherwise, the data files are not useful. Thus, to update data in the file, we need to update the data files with latest information. To update a particular record of the data file, the data may be stored anywhere in the file; it is necessary to obtain the location (in terms of byte number) at which the data object is stored.

The `sizeof()` operator determines the size of the object. Consider the following statements:

```
(a) int size = sizeof(o);
```

Here, `o` is an object, and `size` is an integer variable. The `sizeof()` operator returns the size of the object `o` in bytes, and it is stored in the variable `size`. Here, one object is equal to one record.

The position of the `n`th record or object can be obtained using the following statement:

```
(b) int p = (n-1 * size);
```

Here, `p` is the exact byte number of the object that is to be updated; `n` is the number of the object; and `size` is the size in bytes of an individual object (record).

Suppose we want to update the fifth record. The size of the individual object is 26.

```
(c) p = (5-1*26) i.e. p = 104
```

Thus, the `fifth` object is stored in a series of bytes from `105 to 130`. Using `seekg()` and `seekp()` functions, we can set the file pointer at that position.

16.19 Write a program to create a text file. Add and modify records in the text file. The record should contain name, age, and height of a boy.

```
#include<stdio.h>

#include<process.h>

#include<fstream.h>

#include<conio.h>


class boys

{

char name [20];

int age;

float height;

public:

void input()

{

cout<< “Name:”; cin>>name;

cout<< “Age:”; cin>>age;

cout<< “Height:”; cin>>height;

}

void show (int r)

{

cout<<“\n”<<r<<“\t”<<name<<“\t”<<age <<“\t”<<height; }

};
```

```
boys b[3];
fstream out;
void main()
{
clrscr();
void menu (void);
out.open (“boys.doc”, ios::in | ios::out | ios::noreplace);
menu();
}
void menu(void)
{
void get(void);
void put(void);
void update(void);
int x;
clrscr();
cout<<“\n Use UP arrow key for selection”;
char ch=‘ ’;
gotoxy(1,3);
printf (“ADD()”);
gotoxy(1,4);
printf (“ALTER()”);
gotoxy(1,5);
```

```
printf (“EXIT()”);
x=3;
gotoxy(7,x);
printf (“*”);
while (ch!=13)
{
ch=getch();
if (ch==72)
{
if (x>4)
{
gotoxy(7,x);
printf (“ ”);
x=2;
}
gotoxy(7,x);
printf (“ ”);
gotoxy(7,++x);
printf (“*”);
}
}
switch(x)
{
```

```
case 3 : get(); put(); getche(); break;

case 4 : put(); update(); put(); getche(); break;

default : exit(1);

}

menu();

}

void get()

{

cout<<“\n\n\n\n Enter following information:\n”;

for (int i=0;i<3;i++)

{

  b[i].input();

out.write ((char*) & b[i],sizeof(b[i]));

}

}

void put()

{

out.seekg(0,ios::beg);

cout<<“\n\n\n Entered information \n”;

cout<<“Sr.no Name Age Height”;

for (int i=0;i<3;i++)

{

out.read((char *) & b[i],
sizeof(b[i]));

b[i].show(i+1);

}

}

void update()

{

int r, s=sizeof(b[0]);

out.seekg(0,ios::beg);

cout<<“\n”<<“Enter record no. to update:”;

cin>>r;

r=(r-1)*s;

out.seekg(r,ios::beg);

b[0].input();

out.write ((char*) & b[0],sizeof(b[0]));

put();

}
```

OUTPUT

Use UP arrow key for selection

ADD (*)

ALTER()

EXIT()

```
Enter following information :

Name : Sachin

Age : 28

Height : 5.4

Name : Rahul

Age : 28

Height : 5.5

Name : SauravAge : 29

Height : 5.4

Entered information

Sr.no Name Age Height

1 Sachin 28 5.4

2 Rahul 28 5.5

3 Saurav 29 5.4
```

Explanation: In the above program, the `class boys` contains the data member's name, age, and height. The `class boys` also contains the member functions `input()` and `show()`. The `input()` function is used to read data, and the `show()` function displays data on the screen.

After class definition and before the `main()` function array of objects, `b[3]` and `fstream` object `out` are declared. They are declared before `main()` for global access. The file `boys.doc` is opened in the input and output modes to perform both read and write operations.

The `menu()` function displays the menu on the screen. The menu items can be selected using the up arrow key. Hit enter to start the operation. There are another three user-defined functions. They are `get()`, `put()`, and `update()`. The `get()` function calls the member function `input()` to read data through the keyboard. The `get()` function writes the data using the `write()` function. The `put()` function calls the member function `show()`. The `put()` function calls the member function `show()`. The `put()` function reads the data from the file using the `read()` function.

The `update()` function is used to modify the previous record. The `seekg()` function sets the file pointer at the beginning of the file. The `sizeof()` operator determines the size of the object and stores it in the variable `s`.

16.12 ERROR HANDLING FUNCTIONS

Until now, we have performed the file operation without any knowledge of the failure or success of the function `open()` that opens the file. There are many reasons; they may result in errors during read/write operations of the program.

9. An attempt to read a file that does not exist
10. The file name specified for opening a new file may already exist
11. An attempt to read the contents of a file when the file pointer is at the end of the file
12. Insufficient disk space
13. Invalid file name specified by the programmer
14. An effort to write data to the file that is opened in the read-only mode
15. A file opened may be already opened by another program
16. An attempt to open the read-only file for writing operation
17. Device error

The `stream state` member from the `class ios` receives values from the status bit of the active file. The class `ios` also contains many different member functions. These functions read the status bit of the file where an error occurred during program execution are stored. These functions are depicted in Table 16.5, and various status bits are described in Table 16.4.

All streams such as `ofstream, ifstream,` and `fstream` contain the state connected with them. Faults and illegal conditions are managed (controlled) by setting and checking the state properly. Figure 16.13 describes it more clearly.

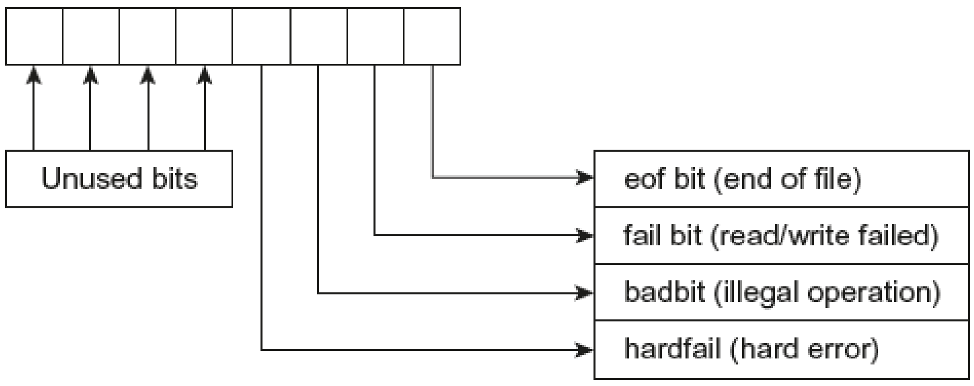


Fig. 16.13 Status bits

Table 16.4 Status Bits

eofbit	End of file encountered.	0x01
failbit	Operation unsuccessful	0x02
badbit	Illegal operation due to wrong size of buffer	0x04
hardfail	Critical error	0x08

Table 16.5 Error trapping functions

Functions	**Working and return value**
fail()	Returns non-zero value if an operation is unsuccessful. This is carried out by reading the bits `ios::failbit`, `ios:: badbit`, and `ios::hardfail` of `ios::state.`
eof()	Returns non-zero value when the end of the file is detected; otherwise, it returns zero. The ios`::eofbit` is checked.
bad()	Returns non-zero value when an error is found in the operation. The `ios::badbit` is checked.
good()	Returns non-zero value if no error occurred during the file operation, that is, no status bits were set. This also indicates that the above functions are false. When this function returns true, we can proceed with the file operation.
rdstate()	Returns the stream state. It returns the value of various bits of the `ios::state.`

The following examples illustrate the techniques of error checking:

1. **An attempt to open a non-existent file for reading**

```
ifstream in(data.txt”);

if (!in){ cout<< File not found”; }
```

In the above format, an attempt is made to open a file for reading. If the file already exists, it will be opened; otherwise, the operation fails. Thus, by checking the value of the object `in`**,** we can confirm the failure or success of the operation and according to this, further processing can be decided.

2. **An attempt to open a read-only file for writing**

```
ofstream out(data.txt”);

if (!out)
```

```
cout<<Unable to open file”;

else

cout<<“File opened”;
```

Suppose the data.txt file is protected (marked read only) or used by another application in a multitasking operating environment. If the same file is opened in the write mode as shown above, the operation fails. By checking the value of the object out with the `if()` statement, we can catch the error and transfer the program control to a suitable sub-routine.

3. **Checking end of file**

```
ifstream in(data.txt”);

while (!in.eof())

{

// read data from file

// display on screen

}
```

We may seek to open an existing file and read its contents. After opening a file in the read mode, it is necessary to read the characters from the file using an appropriate function (`read()` or `get()`). While reading a file, the get pointer is advanced to the successive characters, and the same process can be repeated using loops. The compiler cannot determine the end of the file. The `eof()` function determines the end of the file. Thus, by checking the value of the `eof()` function, we can determine the end of the file. In addition, by checking the value of the object, the end of the file is determined. Such conditions should be placed in the `while` loop parentheses. `While` reading the file, use only `while` or `for` loop.

4. **Illegal file name**

```
ifstream in(“*+**”);

while (!in.eof())
```

```
{
// read data from file
// display on screen
}
```

While performing a file operation, it is the user's responsibility to specify the correct file name. If an illegal file name is specified by the user, the file operation fails. In the above format, "*+**" is given as a file name that is invalid.

5. **Operation with unopened file**

```
ifstream in("DATA");
while (!in.eof())
{
// read data from file
// display on screen
}
```

Suppose the "DATA" file does not exist and an attempt is made to open it for reading. Any operation applied with this file will be of no use. Hence, while performing a file operation, first we have to check whether the file is successfully opened or not. After the confirmation, we can proceed to the next step.

Programs referred to in the above discussion are explained below.

16.20 Write a program to detect whether the file is opened successfully or not.

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

```
#include<string.h>

ifstream in; // Global object


void main()

{

clrscr();

void show (void);

in.open(“dat”) ;

char c;

if (in!=0) show();

else

cout<<“\n File not found”;

}

void show()

{

char c;

cout<<“\n Contents of file:”;

while (in)

{

in.get(c);

cout<<c;

}

}
```

OUTPUT

File not found

Explanation: In the above program, the object `in` of the ifstream class is declared globally. It can be accessed by any normal function. In the `main()` function, `in` is the object used for opening the file. If the `open()` function fails to open the file, it returns a zero; otherwise, it returns a non-zero value. The `if` statement checks the value of the object `in`, and if it is a non-zero `show()` function, it is invoked; otherwise, “File not found” message is displayed. The `show()` function reads the file and displays the contents on the screen. In the above program, the `open()` function tries to open “dat” file, which does not exist. Hence, the output is “File not found.” If the specified file exists, the contents of the file will be displayed.

16.21 Write a program to display status of various errors trapping functions.

```
#include<fstream.h>

#include<conio.h>


void main()

{

clrscr();

ifstream in;

in.open (“text.txt”, ios::nocreate);

if (!in)

cout<<“\n File not found”;

else
cout<<“\nFile=”<<in;

cout<<“\nError state=”<<in.rdstate();

cout<<“\ngood()=”<<in.good();

cout<<“\neof()=”<<in.eof();

cout<<“\nfail()=”<<in.fail();

cout<<“\nbad()=”<<in.bad();

in.close();

}
```

OUTPUT

```
File not found

Error state = 4

good() = 0

eof() = 0

fail() = 4

bad() = 4
```

Explanation: In the above program, an attempt is made to open a non-existent file. The `if` statement checks the value of the object `in`. The specified file does not exist; hence, it displays the message “File not found.” The program also displays the values of various bits using the functions `good()`, `eof()`, `bad()`, `fail()`, and `rdstate()` error trapping functions. For more information about these functions, please refer Table 16.4.

16.13 COMMAND-LINE ARGUMENTS

An executable program that performs a specific task for the operating system is called a `command`. The commands are issued from the command prompt of the operating system. Some arguments are associated with the commands; hence these arguments are called `command-line arguments`. These associated arguments are passed to programs.

Similar to C, in C++, every program starts with a `main()` function, and this function marks the beginning of the program. We have not provided any arguments so far in the `main()` function. Here, we can make arguments in the main function as in other functions. The `main()` function can receive two arguments, and they are (1) `argc` (`argument counter`) and (2) `argv` (`argument vector`). The first argument contains the number of arguments, and the second argument is an array of char pointers. The `*argv` points to the command-line arguments. The size of the array is equal to the value counted by the `argc`. The information contained in the command line is passed on to the program through these arguments when the `main()` is called up by the system.

1. **Argument argc:_**The argument `argc` counts the total number of arguments passed from command prompt. It returns a value that is equal to the total number of arguments passed through the `main()`.
2. **Argument argv:_**It is a pointer to an array of character strings that contains names of arguments. Each word is an argument.

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

Example - ren file1 file2.
```

Here, `file1` and `file2` are arguments, and copy is a command. The first argument is always an executable program followed by associated arguments. If you do not specify the argument, the first program name itself is an argument but the program will not run properly and will flag an error. The contents of `argv[]` would be as follows:

```
argv [0] → ren

argv [1] → file1

argv [2] → file2
```

16.22 Write a program to simulate rename command using command line arguments.

```
#include<stdio.h>
```

```
#include<fstream.h>

#include<conio.h>

#include<process.h>


main(int argc, char *argv[])

{

fstream out;

ifstream in;

if (argc<3 )

{

cout<<“Insufficient Arguments”;

exit(1);

}

in.open(argv[1],ios::in | ios::nocreate);

if (in.fail())

{

cout<<“\nFile Not Found”;

exit(0);

}

in.close();

out.open(argv[2],ios::in | ios::nocreate);

if (out.fail())

{ rename(argv[1],argv[2]); }
else

cout<<“\nDuplicate file name or file is in use.”;

return 0;

}
```

Explanation: In the above program, the `main()` receives two file names. The existence of the file can be checked by opening it in the read mode. If the file does not exist, the program is terminated. On the other hand, if the second file exists, the renaming operation cannot be performed. The renaming operation is carried out only when the first file exists and the second file does not exist. Make exe file of this program and use it on the command prompt.

16.14 STRSTREAMS

ostrstream

The `strstream` class is derived from the `istrstream` and `ostrstream` classes. The `strstream` class works with the memory. Using the object of the `ostrstream` class, different types of data values can be stored in an array.

16.23 Write a program to demonstrate use of ostrstreams object.

```
#include<strstream.h>

#include<iomanip.h>

#include<conio.h>


main()

{

clrscr();

char h=‘C’;

int j=451;
float PI=3.14152;

char txt[]=”applications”;

char buff[70];

ostrstream o (buff,70);

o<<endl << setw(9)<<“h=”<<h<<endl <<setw(9)<<“j=”<<oct<<j<<endl

<<setw(10)<<“PI=”<< setiosflags(ios::fixed)<<PI<<endl<<setw(11)

<<“txt=”<<txt <<ends;

cout<<o.rdbuf();

return 0;

}
```

OUTPUT

h=C

j=703

PI=3.14152

txt= applications

Explanation: The `strstream` deals with the memory. If we want to pick characters from an `strstream` or we want to add characters into the `strstream,` this can be done by creating `istrstream` and `ostrstream` objects. When the object `o` is created, the constructor of the `ostrstream` is executed. Once an object of the `ostrstream` is created, we can assign any formatted text to the array associated with it. The statement `cout<<o.rdbuf()` displays the formatted information on the screen.

istrstream

It is one of the base classes of the `strstream` class. Using the object of the `istrstream` class, data can be extracted from an array. Suppose a character array contains numbers and characters. You can extract a number from an array and assign it to an integer variable. Similarly, other values can be extracted from an array. The following program illustrates this:

16.24 Write a program to demonstrate the use of istrstream.

```
#include<strstream.h>

#include<conio.h>


main()

{

clrscr();

char *book;

int pages;

float price;

char *text=”550 175.75 C++”;

istrstream o(text);

o>>pages>>price>>book;

cout<<endl <<pages <<endl <<price<<endl<book;

cout<<o.rdbuf();

return 0;

}
```

OUTPUT

550

175.75

C++

Explanation: The `istrstreams` is the opposite of the `strstream.` It picks different types of data from an array. In the above program, a character pointer text contains the data of `integer, float,` and `character` type. Using the object of the `istrstreams` class, we can separate the contents and store them in appropriate variables. The `*book` pointer variable displays a string. The remaining contents are displayed by the function `rdbuf()`.

16.15 SENDING OUTPUT TO DEVICES

It is also possible to send information of files directly to devices such as a printer or monitor. Table 16.6 describes various devices along with their names and descriptions. The following program illustrates the use of such devices in the program:

Table 16.6 Standard Devices

Device Name	Description
CON	Console (monitor screen)
COM1 or AUX	Serial port – I
COM2	Serial port – II
LPT1 OR PRN	Parallel printer – I
LPT2	Parallel printer – II
LPT3	Parallel printer – III
NUL	Dummy device

16.25 Write a program to read a file and sent data to the printer.

```
#include<fstream.h>

#include<iostream.h>

#include<conio.h>
```

```
#include<process.h>

#define eject out.put(‘\x0C’);


void main()

{

clrscr();

char h;

char name[20];

cout<<“Enter file name:”;

cin>> name;

ifstream in (name);

if (!in)

{

cerr <<endl<<“File opening error”;

_cexit();

}

ofstream out (“LPT1”);

if(!out)

{

cerr <<endl<<“device opening error”;

_cexit();

}

while (in.get(h)!=0)
out.put(h);

eject;

}
```

Explanation: In the above program, the file name is entered through the keyboard, and it is opened for reading purpose. The `ifstream` object `in` opens the file. The `ofstream` object `out` activates the printer. The `if` statements check both the objects for detecting operation status, that is, whether the operations have failed or are successful. The `while` loop reads data from the file and using the `put()` statement, it passes it to the devices associated with the object `out`. In this program, the data read are passed to the printer. The macro `eject` defined at the beginning of the program advances the page of the printer. In case the printer is not attached, the message displayed will be as given below.

```
Error Message

System Message

Error accessing LPT1 device

» Retry « Cancel
```

The user can select `retry` if he or she had attached the printer; otherwise, by selecting `cancel`, the operation can be cancelled.

16.16 MORE PROGRAMS

16.26 Write a program to copy contents of one file to another file.

```
#include<fstream.h>

#include<conio.h>

#include<process.h>


main()

{
```

```
clrscr();

char s[12],t[12],c;

fstream out;

ifstream in;

cout<<“\n Enter a Source file name:”;

cin>>s;

cout<<“\n Enter a target file name:”;

cin >>t;

in.open(s,ios::in | ios::nocreate);

if (in.fail())

{

cout<<“\nFile “<<s <<“ Not Found”;

exit(1);

}

out.open(t,ios::out | ios::nocreate);

if (out.fail())

{

out.open(t,ios::out);

while (in.eof()==0)

{

in.get(c);

out.put(c);

}
}

else

cout<<“\n Target file already exist.”;

in.close();

out.close();

return 0;

}
```

OUTPUT

Enter a Source file name : DATA

Enter a target file name : TEXT

Explanation: In the above program, the user enters `source` and `target` file names. The existence of the files is checked. In case the `source` file is absent and the `target` file is already present, the copying of data will not take place. Appropriate messages are displayed when the file names are not properly entered.

The `target` file is opened in read and `nocreate` mode. The `nocreate` flag prevents the opening of a new file if the file is absent. If it is unsuccessful, then the `open()` statement within the `if` statement opens the file for writing. The `while` loop executes the function till the file pointer reaches the end of the source file. The `get()` statement reads data from the source file, and the `put()` statement writes the read data to the target file. Thus, the copying of data is carried out. After termination of the `while` loop, both the files are closed using `close()` functions.

16.27 Write a program to copy content of one file in another file in reverse order. Display the contents of the screen.

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

```
#include<conio.h>

#include<process.h>


main()

{

clrscr();

char s[12],t[12],c;

fstream out;

ifstream in;

cout<<“\n Enter a Source file name:”;

cin>>s;

cout<<“\n Enter a target file name:”;

cin >>t;

in.open(s,ios::in ); //| ios::nocreate);

if (in.fail())

{

cout<<“\nFile”<<s <<“ Not Found”;

exit(1);

}

else

in.seekg(0,ios::end);

out.open(t,ios::out | ios::nocreate);

int b;
```

```
if (out.fail())

{

out.open(t,ios::out);

in.seekg(0,ios::end);

b=in.tellg();

for (int i=1;i<=b;i++)

{

in.seekg(-i,ios::end);

in.get(c);

out.put(c);

cout<<c;

}

}

else

cout<<“\nTarget file alredy exist.”;


in.close();

out.close();

return 0;

}
```

OUTPUT

Enter a Source file name : cpp

Enter a target file name : cp2

GnimmargorP detneirO tcejbO

Explanation: In the above program, `source` and `target` file names are entered. The content of the `source` file is copied to the `target` file in reverse order. The `source` file is opened for reading, and the `target` file is opened for writing. Using the `tellg()` function, the size of the source file is obtained and stored in the variable `b`. The `for` loop executes from `1 to b` (size of source file). The `seekg()` function moves the get file pointer in the reverse order, that is, from end to top. The argument `–i` specifies the number of bytes to be read from the end of the file. The character read by the `get()` function is written to the target file by the `put()` function. The `cout()` statement displays the contents of the variable `c` on the screen.

16.28 Write a program to open a file in read and write mode. Write data to the file and read from it.

```
#include<fstream.h>

#include<conio.h>


void main()

{

clrscr();

ofstream out (“data.txt”); // creates file for writting

char name[]=”SANJAY”;

int age=25;

float ht=4.5;

out <<name<<“\t”<<age<<“\t”<<ht; // writes data to the data.txt

out.close(); // closes file
ifstream in (“data.txt”); // opens file for reading

in>>name >>age>>ht; // reads data from file and assigns to
variables

cout<<endl<<“Name:”<<name; // display data on the screen

cout<<endl<<“Age:”<<age;

cout<<endl<<“Height:”<<ht;

}
```

OUTPUT

Name : SANJAY

Age : 25

Height : 4.5

Explanation: In the above program, the file `“data.txt”` is opened in the write mode. The values of the variable’s name, age, and ht are written to the file. The file is closed using the `close()` function.

Again, the same file is opened in the read mode. The data read are assigned to respective variables and displayed on the screen using the `cout()` statement.

16.29 Write a program to write data to the file in string format also read and display the data in the same fashion.

```
#include<fstream.h>

#include<conio.h>
```

```
void main()

{

clrscr();

char text[100];

ofstream out (“data.txt”);

out<<“ Programming with ANSI and Turbo C”;

out<<“\n Teaches you C with practical programs”;

out.close();

ifstream in (“data.txt”);

while (!in.eof())

{

in.getline(text,100);

cout<<endl<<text;

}

}
```

OUTPUT

Programming with ANSI and Turbo C

Teaches you C with practical programs

Explanation: This program is similar to the previous one. Here, a string is written to the file. Using the `getline()` function, the string is read from the file and displayed. As soon as the end of the file is detected, the `while` loop terminates.

16.30 Write a program to copy contents of one file to another file. Use `rdbuf()` function.

```
#include<fstream.h>

#include<conio.h>


void main()

{

clrscr();

char sfile[20], dfile[20];

cout<<“\nEnter source file:”;

cin>>sfile;

cout<<“\nEnter destination file:”;

cin>>dfile;

ifstream in(sfile);

ofstream out(dfile);

out<<in.rdbuf();

in.close();

out.close();

}
```

OUTPUT

Enter source file : data.txt

Enter destination file : c.txt

Explanation: In the above program, two character arrays `sfile [20]` and `dfile [20]` are declared to hold source and destination file names, respectively. The `rdbuf()` copies entire the `source` file to the `target` file.

16.31 Write a program to display strings and their addresses.

```
#include<strstream.h>

#include<conio.h>


main()

{

clrscr();

char s[]=”Sunday”;

char *r=”Monday”;

cout<<“Strings”<<endl;

cout<<s <<endl;

cout<<r <<endl;

cout<<endl<<“Addresses”<<endl;

cout<<&s<<endl; // c style

cout<<(void*) r<<endl;

cout<<(unsigned)&s<<endl;

return 0;

}
```

OUTPUT

```
Strings
Sunday
Monday
Addresses
0x8f9cffee
0x8f9c00b1
65518
```

Explanation: In the above program, character array `s[]` and character pointer `r` are initialized with strings. The strings and their memory addresses are displayed using `cout()` statements. The address can be displayed using the & operator in traditional C style. We can also display the address as per the statement `cout<<(void*) r`. The address is converted from `char*` into `void*`. The statement `cout<<(unsigned)&s` converts the address into an unsigned integer and displays it.

16.32 Write a program to show errors occurring during file opening operations.

```
#include<fstream.h>

#include<conio.h>

#include<process.h>


void main()

{
```

```
clrscr();
void errors (ofstream &);
ofstream f;
f.open (“data.txt”,ios::noreplace);
if (!f) errors(f);
else f <<“Hope is a walking dream”;
f.close();
}
void errors (ofstream &f)
{
cout<<endl<<“File opening errors”;
cout<<endl <<“Error state=”<<f.rdstate();
cout<<endl<<“fail()=”<<f.fail();
cout<<endl<<“eof()=”<<f.eof();
cout<<endl<<“bad()=”<<f.bad();
cout<<endl<<“good()=”<<f.good();
_cexit();
}
```

OUTPUT

```
File opening errors
Error state = 4
```

fail()=4

eof() = 0

bad()=4

good()=0

Explanation: In the above program, the file `data.txt` is in the write mode. If the file opening operation fails, the function `error()` is executed. The `if` statement checks the value of the object `f`, and if it is zero, then the function `errors()` are invoked, which display the errors. The error states displayed in the output are illustrated in Table 16.7:

Table 16.7 Return values of functions

rdstate() (error state = 4)	The rdstate() function gives the value 4, and it points that the file operation was unsuccessful.
fail()=4 & bad()=4	These functions display a non-zero value, and this is due to an error that is generated during the operation.
eof() = 0	This function returns zero, because the file pointer is not at the end of the file.
good() = 0	This function returns zero, because no bit sets.

16.33 Write a program to enter numbers using command line arguments. Calculate the product of all the numbers.

```
#include<strstream.h>

#include<conio.h>


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

{

int k=1;
long n,s=1;
if (argc<2)
{
cout<<“ Enter numbers ”;
return ;
}
while (--argc)
{
istrstream value (argv[k]);
value>>n;
s*=n;
k++;
}
cout<<endl<<“ Multiplication of entered numbers:”<<s<<endl;
}
```

OUTPUT

```
C:\tc3>cmd 4 4 4
Multiplication of entered numbers : 64
C:\tc3>cmd 45545 2
Multiplication of entered numbers : 91090
```

Explanation: The above program can be executed on command prompt by creating its exe file. The number of arguments entered by the user is checked by the first `if` statement; if arguments are less than two, the message displayed will be `“Enter numbers.”`

If the user enters numbers followed by the file name, the product of all the numbers is calculated and displayed. The object `value` is the object of the class `istrstream`, and it is connected to the buffer. Consider the statement `value>>n`; here, the object `value` takes data from the buffer and assigns it to the variable `n`.

SUMMARY

1. Nowadays, a huge amount of data is processed in the computer networking. The information is uploaded or downloaded from the desktop computer. The information transfer in computer networking in day-today life is done in the form of files. The data are saved in the file on the disk. The file is an accumulation of data stored on the disk.
2. A stream is nothing but the flow of data. In the object-oriented programming, the streams are controlled using the classes.
3. The `istream` and `ostream` classes control the input and output functions, respectively.
4. The `iostream` class is also a derived class. It is derived from the `istream` and `ostream` classes. There are another three useful derived classes. They are `istream_withassign, ostream_withassign,` and `iostream_withassign`, and they are derived from `istream, ostream,` and `iostream,` respectively.
5. `filebuf` accomplishes input and output operations with the files. The `fstreambase` acts as a base class for `fstream`, `ifstream,` and `ofstream.` The `ifstream` class is derived from the `fstreambase` and `istream` classes by multiple inheritance. It can access the member functions such as `get()`, `getline()`, `seekg()`, `tellg(),` and `read()`. The `ofstream` class is derived from the `fstreambase` and `ostream` classes. It can access the member functions such as `put()`, `seekp()`, `write()`, and `tellp().` The `fstream` class allows for simultaneous input and output on a `filebuf`. The member function of the `istream` and `ostream` in the base classes starts the input and output, respectively.
6. The following two methods are used for the opening of a file: (A) Constructor of the class; (b) Member function `open()` of the class.
7. The class `ofstream` creates output stream objects, and the class `ifstream` creates input stream objects.
8. The `close()` member function closes the file.
9. The `open()` function uses the same stream object. The `open()` function has two arguments. The first is the file name, and the second is the mode.
10. When the end of the file is detected, the process of reading data can be easily terminated. The `eof() function()` is used for this purpose. The `eof()` stands for the end of the file. It is an instruction given to the program by the operating system that the end of the file is reached. The `eof()` function returns `one` when the end of the file is detected.

11. The mode `ios::out` and `ios::trunc` are near about same. The `ios:: app` lets the user add data at the end of the file, whereas `ios: ate` allows the user to add or update data anywhere in the file.
12. The `seekg()` function shifts the associated file's input (get) file pointer. The `seekp()` function shifts the associated file's output (put) file pointer.
13. `seekg()` -- Shifts input (get) pointer to a given location
14. `seekp()` -- Shifts output (put) pointer to a given location
15. `tellg()` -- Provides the present position of the input pointer
16. `tellp()` -- Provides the present position of the output pointer
17. The `put()` and `get()` functions are used to read or write a single character, whereas the `write()` and `read()` functions are used to read or write blocks of binary data.
18. The `fail(), eof(), bad(),` and `good()` are error trapping functions.
19. An executable program that performs a specific task for the operating system is called a `command`. The commands are issued from the command prompt of the operating system. Some arguments are to be associated with the commands; hence, these arguments are called `command-line arguments`.
20. Syntax - main (int argc, char * argv[]);

EXERCISES

(A) Answer the following questions

1. What is a stream?
2. What is a file?
3. Describe the different classes derived from `ios` that control the disk I/O operations.
4. Describe the two methods of opening a file.
5. Explain the detection of the end of a file with the function `eof()`.
6. Describe the syntax of the `open()` function with its arguments.
7. What are the different types of file opening modes? List their names along with their meanings.
8. Describe file manipulators with their syntaxes.
9. Describe the various error trapping functions.
10. What are the possible reasons for the failure of the `open()` function?
11. What are command-line arguments?
12. Explain the use of the NOT operator and `eof()` function.
13. Explain sequential and random file operations.
14. How would you write data in a file in binary format?
15. What are the limitations of using the `put()` and `get()` functions?
16. Explain the uses of the `ostrstream` and `istrstream` classes.
17. Explain the differences between binary and ASCII files.

(B) Answer the following by selecting the appropriate option

1. The `eof()` stands for

    1. **end of the file**
    2. error opening the file
    3. error of the file
    4. none of the above
2. Command-line arguments are used with function
    1. **`main()`**
    2. member function
    3. with all functions
    4. none of the above
3. The statement in.seekg(0,ios::end) sets the file pointer
    1. **at the end of the file**

Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a technique that allows a single function or class to work with different data types. Using a template, we can create a single function or a class that can process any type of data; that is, the formal arguments of a template are of template (generic) type. They can accept data of any type, such as int, float, and long. Thus, a single function or class can be used to accept values of a different data type. In general templates are used to create a family of classes or functions.

Cass Templates

In order to declare a class of template type, the following syntax is used:

```
template < class T>
class name_of_class
{
        // class data member and function
}
```

The first statement template < class T> tells the compiler that the following class declaration can use the template data type. The T is a variable of template type that can be used in the class to define a variable of template type. Both template and class are keywords. The <> (angle bracket) is used to declare the variables of template type that can be used inside the class to define the variables of template type. One or more variables can be declared separated by a comma. Templates cannot be declared inside classes or inside functions. They should be global and should not be local.

/*Write a C++ program to show values of different data types using overloaded constructor. */

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

class data
{
    public:
    data(char c)
    {
                cout<<" c="<<c <<"Size in bytes:"<<sizeof(c)<<endl;
    }
    data(int c)
    {
         cout<<" c="<<c <<"Size in bytes:"<<sizeof(c)<<endl;
    }
    data(double c)
    {
         cout<<" c="<<c <<"Size in bytes:"<<sizeof(c)<<endl;
    }
};
int main()
{
    data h('A'); // passes character type data
    data i(100); // passes integer type data
    data j(68.22); // passes double type data
    return 0;
}
```

Output:

```
c=A     Size in bytes:1
c=100   Size in bytes:4
c=68.22 Size in bytes:8
```

Explanation: In the above program, the class data contains three overloaded one-argument constructors. The constructor is overloaded for char, int, and double type. In function main(), three objects h, i, and j are created, and the values passed are of different types. type. This approach has the following disadvantages:

1. Re-defining the functions separately for each data type increases the source code and requires more time.
2. The program size is increased. Hence, more disk space is occupied.
3. If the function contains a bug, it should be corrected in every function.

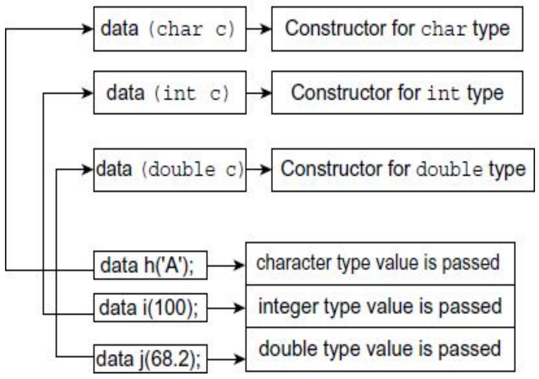

data (char c)
Constructor for char type
data (int c)
Constructor for int type
data (double c)
Constructor for double type
data h('A');
data i(100);
data j(68.2);
character type value is passed
integer type value is passed
double type value is passed

Fig. Working of non-template function

From the above program, it is clear that for each data type we need to define a separate constructor function. According to data, type of argument passed respective constructor is invoked. C++ provides templates to overcome such a problem and helps the programmer develop a generic program. The same program is illustrated with the template as follows:

/* **Write a C++ program to show values of different data types using constructor and template.** */

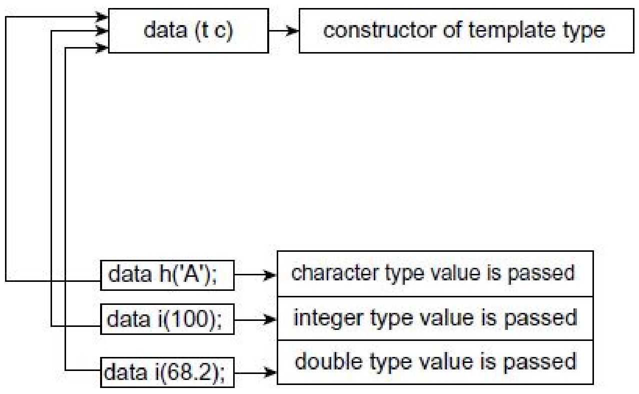
```
#include <iostream>
using namespace std;

template<class T>
class data
{
public:
        data (T c)
        {
                        cout<<" c="<<c <<" Size in bytes:"<<sizeof(c)<<endl;
        }
};
int main()
{
        data <char> h('A');
        data <int> i(100);
        data <float> j(3.12);
        return 0;
}
```

Output:

```
c=A     Size in bytes:1
c=100   Size in bytes:4
c=3.12  Size in bytes:4
```

Explanation: In the above program, the constructor contains a variable of template T. The template class variable can hold values of any data type. While declaring an object, the data type name is given before the object. The variable of template type can accept the values of any data type. Thus, the constructor displays the actual values passed.

Figure shows the working of the program.


data (t c)
constructor of template type
data h('A');
character type value is passed
data i(100);
integer type value is passed
data i(68.2);
double type value is passed

Function Templates

The declaration of a normal template function can be done in the following manner:

```
template < class T>
retun_type function_name (arguments )
{
        // code
}
```

/* Write a C++ program to define normal template function.*/

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

template<class T>
void display(T a, T b)
{
        cout<<a<<" | "<<b<<endl;
}
int main() {

        display(1,2);
        display("Hellow", "Suresh");
        display(1.5,20.5);
        return 0;

}
```

Output:

```
        1 | 2
        Hellow | Suresh
        1.5 | 20.5
```

Explanation: Before the body of the function display(), the template argument T is declared. The function display () has one argument T of template type. As explained earlier, the template type variable can accept all types of data. Thus, the normal function show can be used to display values of different data types. In main function, the display() functions are invoked with int, string and double type of values being passed. The same is displayed in the output.

Example:

/* Write a C ++ program to create square() function using template. */

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

template <class T>
T square(T a)
{
        return a*a;
}
int main() {
        cout<<square(4)<<endl;
        cout<<square<float>(2.2)<<endl;
        cout<<square<double>(2.2)<<endl;
        return 0;
}
```

Output:

```
16
4.84
4.84
```

<u>Working of Function Templates:</u> In the last few examples, we have learned how to write a function template that works with all data types. After compilation, the compiler cannot guess with which type of data the template function will work. When the template function is called at that moment, from the type of argument passed to the template function, the compiler identifies the data type. Then, every argument of template type is replaced with the identified data type; this process is called ***instantiating.***

Class Templates with More Parameters

Similar to functions, classes can be declared to handle different data types. Such classes are known as class templates. These classes are of generic type, and member functions of these classes can operate on different data types. The class template may contain one or more parameters of generic data type. The arguments are separated by commas with a template declaration. The declaration is as follows:

```
template <class T1, class T2>
class name_of_ class
{
        // class declarations and definitions
}
```

/* Write a C++ program to define a constructor with multiple template variables. */

```
#include <iostream>
using namespace std;
template<class T1, class T2>
class temp
{
        T1 a;
        T2 b;
public:
        temp(T1 x,T2 y)
        {
                a=x;
                b=y;
        }
        void display()
        {
                cout<<"A and B Values:"<<a<<"\t"<<b<<endl;
        }
};
int main() {
        temp<int,float> t1(5,6.5);
        temp<float,float> t2(3.5,6.5);
        temp<char,float> t3('a',3.3);
        t1.display();
        t2.display();
        t3.display();
        return 0;
}
```

Output:

```
A and B Values: 5     6.5
A and B Values: 3.5   6.5
A and B Values: a     3.3
```

Member Function Templates

In the previous example, the template functions defined were inline, that is, they were defined inside the class. It is also possible to define them outside the class. While defining them outside, the function template should define the function, and the template classes are parameterized by the type argument.

/***Write a C++ program to define definition of member function template outside the class and invoke the function.*/**

```
template<class T>
class data
{
public:
        data (T c);
};
template<class T>
data<T>::data(T c)
{
                        cout<<" c="<<c <<" Size in bytes:"<<sizeof(c)<<endl;
}
int main()
{
        data <char> h('A');
        data <int> i(100);
        data <double> j(3.12);
        return 0;
}
```

Output:

```
c=A      Size in bytes:1
c=100   Size in bytes:4
c=3.12  Size in bytes:8
```

Explanation: In the above program, the constructor is defined outside the class. In such a case, the member function should be preceded by the template name as per the following statements:

```
template<class T>
data<T>::data (T c)
```

The first line defines the template, and the second line indicates the template class type T.

Function Templates with Different or Multiple Parameters

In some situations we may need to use more than one type parameter in a function template. If that ever occurs, then declaring multiple type parameters is actually quite simple. All you need to do is add the extra type to the template prefix, so it looks like this:

```
template<class T1, class T2>
retun_type function_name (T1 var1, T2 var2 )
{
        // some code in here...
}
```

Example:

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

template<class T>
void display(T a, T b)
{
        cout<<a<<" | "<<b<<endl;
}
template<class T,class T1>
void display(T a, T1 b)
{
        cout<<a<<" | "<<b<<endl;
}
int main() {

        display(1,2);
        display("Hellow", "Suresh");
        display(1.5,20.5);
        display("Suresh",1235);
        return 0;

}
```

Output:

```
1 | 2
Hellow | Suresh
1.5 | 20.5
Suresh | 1235
```

Overloading of Template Functions

A template function also supports the overloading mechanism. It can be overloaded by a normal function or a template function. While invoking these functions, an error occurs if no accurate match is met. No implicit conversion is carried out in the parameters of template functions. The compiler observes the following rules for choosing an appropriate function when the program contains overloaded functions:

1. Searches for an accurate match of functions; if found, it is invoked
2. Searches for a template function through which a function that can be invoked with an accurate match can be generated; if found, it is invoked
3. Attempts a normal overloading declaration for the function
4. In case no match is found, an error will be reported

/* Write a C++ program to overload a template function.*/

```
#include<iostream.h>
template <class T>
void show(T c)
{
    cout<<“\n Template variable c=”<<c;
}
void show (int f)
{
    cout<<“\n Integer variable f=”<<f;
 }
 int main()
{
     show(‘C’);
     show(50);
     show(50.25);
     return 0;
}
```

Output:

```
Template variable c=C
 Integer variable f=50
 Template variable c=50.25
```

Explanation: In the above program, the function show() is overloaded. One version contains template arguments, and the other version contains integer variables. In main(), the show() function is invoked thrice with char, int, and float values that are passed. The first call executes the template version of the function show(), the second call executes the integer version of the function show()**,** and the third call again invokes the template version of the function show().

Recursion with Template Functions

Similar to normal function and member function, the template function also supports the recursive execution of itself. The following program illustrates this:

/***Write a C++ program to invoke template function recursively.*/**

```
template <class T, class TT>
T number(T raised, TT exponent)
{
        if (exponent <1)
                return 1;
        else
                return raised * number(raised, exponent -1);
}
int main()
{
        // Testing integers
        cout << "Testing integers: 2 raised to 4 is " << number(2, 4) << endl;
        // Testing doubles
        cout << "Testing doubles: 5.5 raised to 2.2 is " << number(5.5, 2.2) << endl;
        // Testing a double and a integer
        cout << "Testing integers: 5.5 raised to 2 is " << number(5.5, 2) << endl;
        return 0;
}
```

Output:

```
        Testing integers: 2 raised to 4’ is 16
        Testing doubles: 5.5 raised to 2.2 is 30.25
        Testing integers: 5.5 raised to 2 is 30.25
```