Unit 4

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.

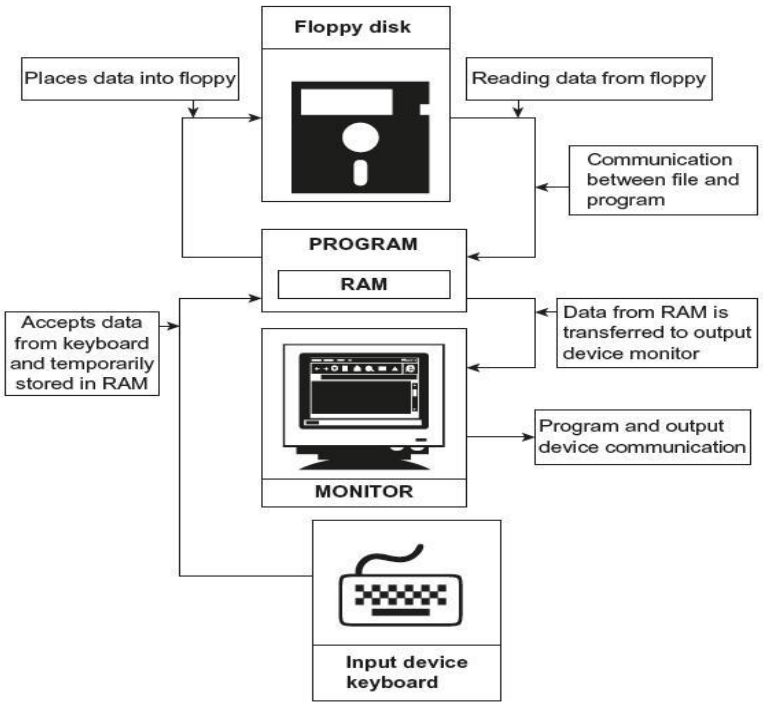


Fig. Communication between program, file, and output device

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.

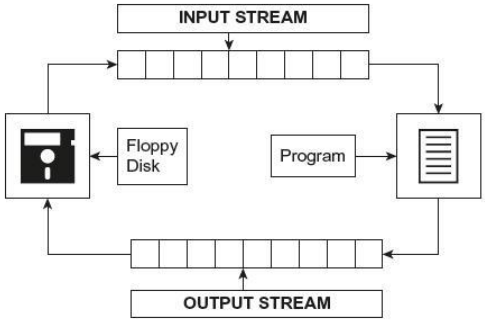
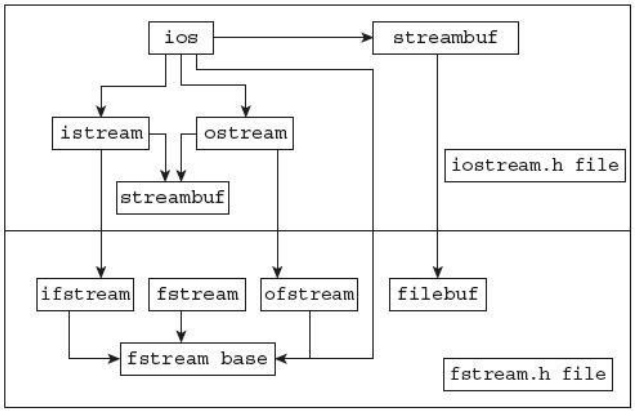


Fig. Input and output streams

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

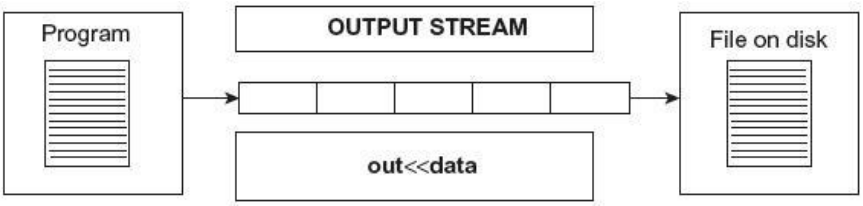
connected with the object in. It is possible to use these file objects in program statements such as stream objects. Consider the following statements:

cout<<“One Two Three”;

The above statement displays the given string on the screen.

out<<“One Two Three”;

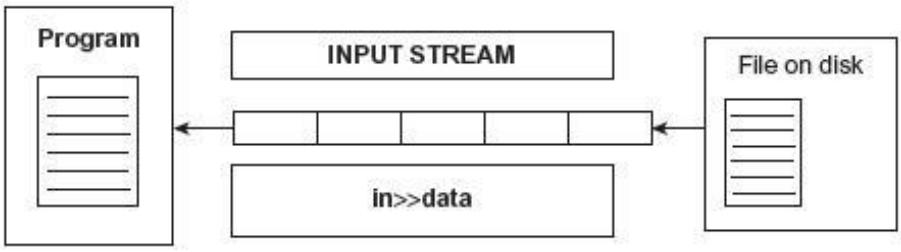
The above statement writes the specified string into the file pointed by the object out as shown in Figure. The insertion operator << has been overloaded appropriately in the ostream class to write data to the appropriate stream.



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 ofstream 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;
}
```

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.

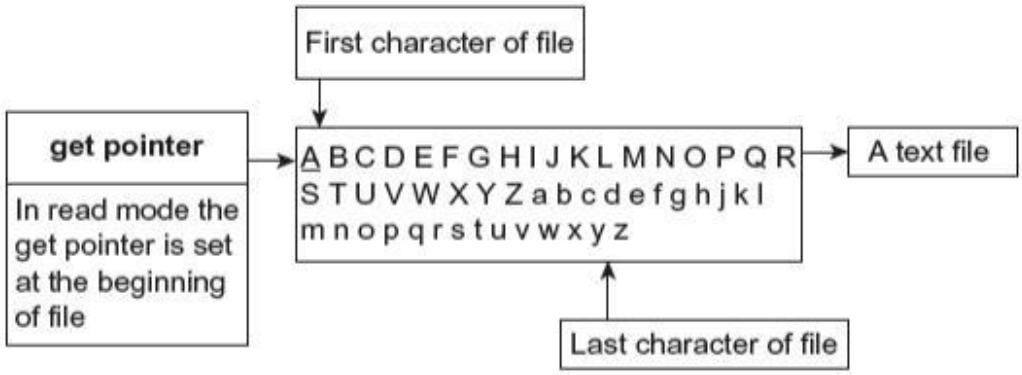


Fig. 16.7 Status of get pointer in read mode

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.

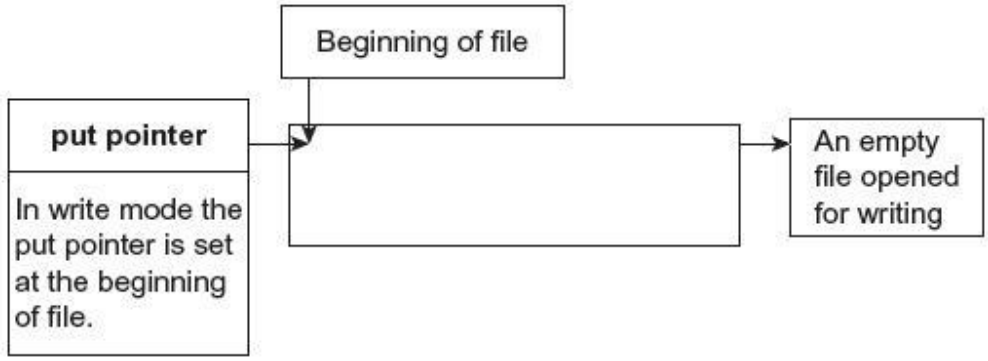


Fig. 16.8 Status of put pointer in write mode

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.

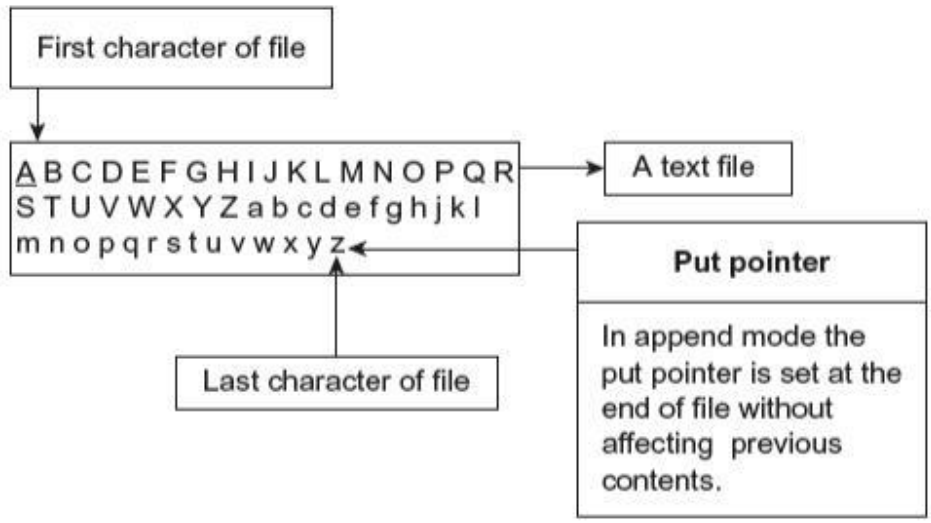


Fig. 16.9 Status of put pointer in append mode

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.

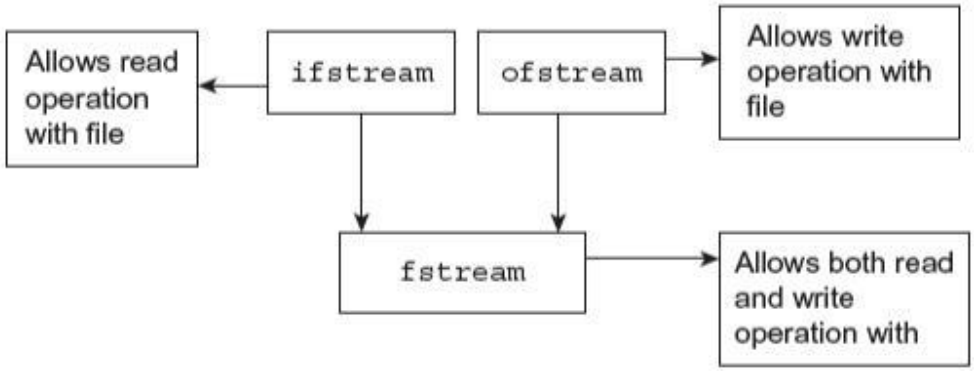


Fig. 16.10 Derivation of fstream class

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
```

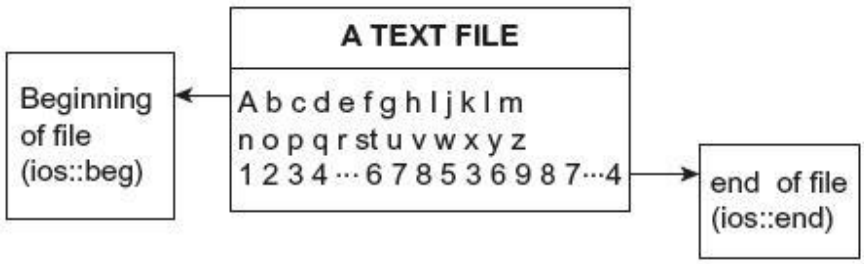


Fig. 16.11 Status of pre-position arguments

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.

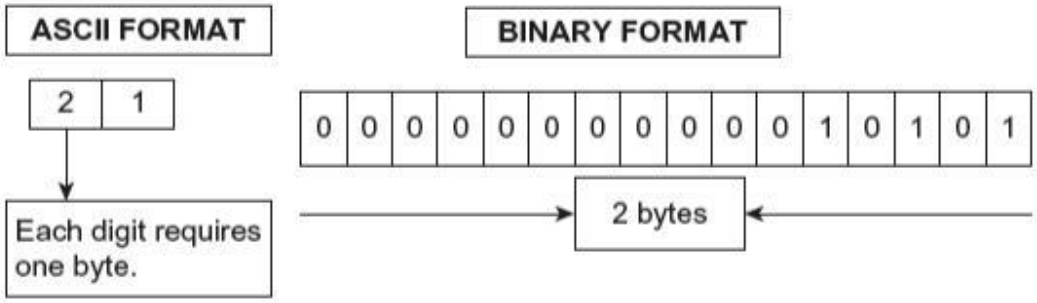


Fig. 16.12 Representation in binary and ASCII formats

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:

808279718265777773787132877384723265788373326578683284858266796 7-1

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.

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.

Exceptions

Exceptions are run time anomalies or unusual condition that a program may encounter during execution.

Examples:

Division by zero

Access to an array outside of its bounds

Running out of memory

Running out of disk space

Principles of Exception Handling: Similar to errors, exceptions are also of two types. They are as follows:

Synchronous exceptions: The exceptions which occur during the program execution due to some fault in the input data.

For example: Errors such as out of range, overflow, division by zero

Asynchronous exceptions: The exceptions caused by events or faults unrelated (external) to the program and beyond the control of the program.

For Example: Key board failures, hardware disk failures

The exception handling mechanism of C++ is designed to handle only synchronous exceptions within a program. The goal of exception handling is to create a routine that detects and sends an exceptional condition in order to execute suitable actions. The routine needs to carry out the following responsibilities:

c) Detect the problem (Hit the exception)
d) Inform that an error has been detected (Throw the exception)
e) Receive error information (Catch the exception)
f) Take corrective action (Handle the exception)

An exception is an object. It is sent from the part of the program where an error occurs to the part of the program that is going to control the error

The Keywords try, throw, and catch

Exception handling mechanism basically builds upon three keywords:

try
catch
throw

The keyword **try** is used to preface a block of statements which may generate exceptions.

Syntax of try statement:

```
try
{
        statement 1;
        statement 2;
}
```

When an exception is detected, it is thrown using a **throw** statement in the try block.

Syntax of throw statement

```
throw (excep);
throw excep;
throw; // re-throwing of an exception
```

A **catch** block defined by the keyword 'catch' catches the exception and handles it appropriately. The catch block that catches an exception must immediately follow the try block that throws the exception

Syntax of catch statement:

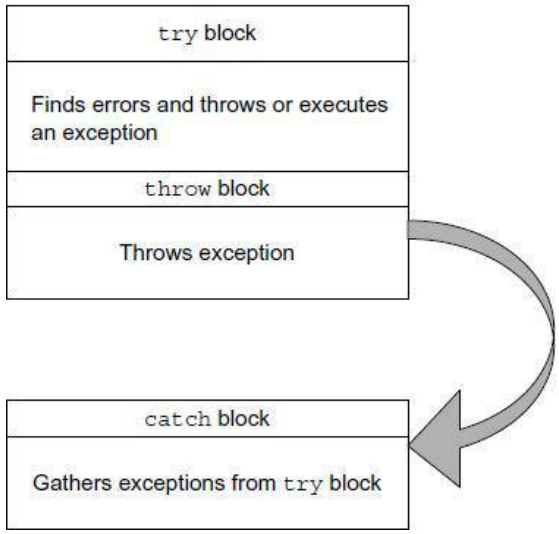
```
try
{
        Statement 1;
        Statement 2;
}
catch ( argument)
{
        statement 3; // Action to be taken
}
```

When an exception is found, the catch block is executed. The catch statement contains an argument of exception type, and it is optional. When an argument is declared, the argument can be used in the catch block. After the execution of the catch block, the

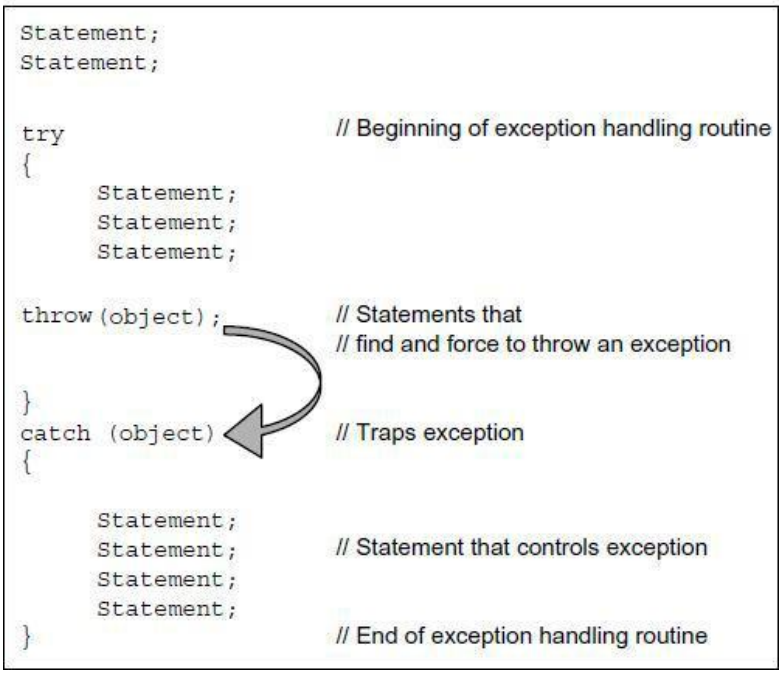
statements inside the blocks are executed. In case no exception is caught, the catch block is ignored, and if a mismatch is found, the program is terminated.

Guidelines for Exception Handling

The C++ exception-handling mechanism provides three keywords; they are try, throw, and catch. The keyword try is used at the starting of the exception. The throw block is present inside the try block. Immediately after the try block, the catch block is present. Figure shows the try, catch, and throw statements.



As soon as an exception is found, the throw statement inside the try block throws an exception (a message for the catch block that an error has occurred in the try block statements). Only errors occurring inside the try block are used to throw exceptions. The catch block receives the exception that is sent by the throw block. The general form of the statement is as per the following figure.



When the try block passes an exception using the throw statement, the control of the program passes to the catch block. The data type used by throw and catch statements should be same; otherwise, the program is aborted using the abort() function, which is executed implicitly by the compiler. When no error is found and no exception is thrown, in such a situation, the catch block is disregarded, and the statement after the catch block is executed.

/* Write a program to illustrate division by zero exception. */

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

int main()
{
        int a, b;
        cout<<"Enter the values of a and b"<<endl;
        cin>>a>>b;
        try{
                if(b!=0)
                        cout<<a/b;
                else
                        throw b;
        }
        catch(int i)
```

```
        {
          cout<<"Division by zero: "<<i<<endl;
        }
        return 0;
}
```

Output:

Enter the values of a and b

2 0

Division by zero: 0

/* Write a program to illustrate array index out of bounds exception. */

```
#include   <iostream>
using namespace std;
int main() {
        int a[5]={1,2,3,4,5},i;
        try{
                i=0;
                while(1){
                        if(i!=5)
                        {
                                cout<<a[i]<<endl;
                                i++;
                        }
                        else
                                throw i;
                }
        }
        catch(int i)
        {
                cout<<"Array Index out of Bounds Exception: "<<i<<endl;
        }
        return 0;
}
```

Output:

1
2
3
4
5
Array Index out of Bounds Exception: 5

/* Write a C++ program to define function that generates exception. */

```
#include<iostream>
using namespace std;
void sqr()
{
        int s;
        cout<<"\n Enter a number:";
        cin>>s;
        if (s>0)
        {
                cout<<"Square="<<s*s;
        }
        else
        {
                throw (s);
        }
}
int main()
{
        try
        {
                sqr();
        }
        catch (int j)
        {
                cout<<"\n Caught the exception \n";
        }
        return 0;

}
```

Ouput:

```
Enter a number:10
Square=100
Enter a number:-1
Caught the exception
```

Multiple catch Statements

It is also possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try (similar to switch statement). The format of multiple catch statements is as follows:

```
 try
{
        // try block
}
catch (type1 arg)
{
        // catch section1
}
catch (type2 arg)
{
        // catch section2
}
. . . . . .
. . . . . .
catch (typen arg)
{
        // catch section-n
}
```

When an exception is thrown, the exception handlers are searched in the order for an appropriate mach. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found the program will terminate.

It is possible that arguments of several catch statements match the type of exception. In such cases, the first handler that matches the exception type is executed.

/*Write a C++ program to throw multiple exceptions and define multiple catch statement. */

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

void num (int k)
{
```

```
        try
        {
                if (k==0) throw k;
                else
                        if (k>0) throw 'P';
                else
                        if (k<0) throw 1.0;
                cout<<"*** end of try block ***\n";
        }
        catch(char g)
        {
                cout<<"Caught a positive value \n";
        }
        catch (int j)
        {
                cout<<"caught an null value \n";
        }
        catch (double f)
        {
                cout<<"Caught a Negative value \n";
        }
        cout<<"*** end of try catch ***\n \n";
}
int main()
{
        cout<<"Demo of Multiple catches"<<endl;
        num(0);
        num(5);
        num(-1);
        return 0;
}
```

Output:

```
Demo of Multiple catches
caught an null value
2     end of try catch ***
 Caught a positive value
3     end of try catch ***
 Caught a Negative value
4     end of try catch ***
 Caught a positive value
5     end of try catch ***
```

Catching Multiple Exceptions

In some situations, we may not able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them. In such circumstances, we can create a catch statement to catch all exceptions instead of a certain type alone.

Syntax:

```
catch(...)
{
     16.10   Statements for handling
     16.11   all exceptions
}
```

/* **Write a C++ program to** `catch` **multiple exceptions.*/**

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

void num (int k)
{
        try
        {
                if (k==0) throw k;
                else
                        if (k>0) throw 'P';
                else
                        if (k<0) throw 1.0;
        }
        catch(...)
        {
                cout<<"Caught an Exception"<<endl;
        }
}
int main()
{
        cout<<"Demo of Multiple catches"<<endl;
        num(0);
        num(5);
        num(-1);
        return 0;

}
```

Output:

```
Demo of Multiple catches
Caught an Exception
Caught an Exception
Caught an Exception
```

Specifying Exceptions

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition. The general form of using an exception specification is:

```
return_type fucntion_name (parameter list) throw (data type list)
{
        // function body
}
```

The data type list indicates the type of exception that is permitted to be thrown. If we want to deny a function from throwing any exception, declaring the data type list void as per the following statement can do this.

```
throw(); // void or vacant list
```

/* Write a C++ program to restrict a function to `throw` only specified type of exceptions. */

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

void check (int k) throw (int)
{
        if (k==1) throw 'k';
        else
                if (k==2) throw k;
        else
                if (k==-2) throw 1.0;
}
int main()
{
        try {
                        check(1);
                        check(-2);
```

```
                check(3);
        }
        catch (char g)
        {
                cout<<"Caught a character exception \n";
        }
        catch (int j)
        {
                cout<<"Caught a character exception \n";
        }
        catch (double s)
        {
                cout<<"Caught a double exception \n";
        }
        cout<<"\n End of main()";
        return 0;
}
```