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 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.

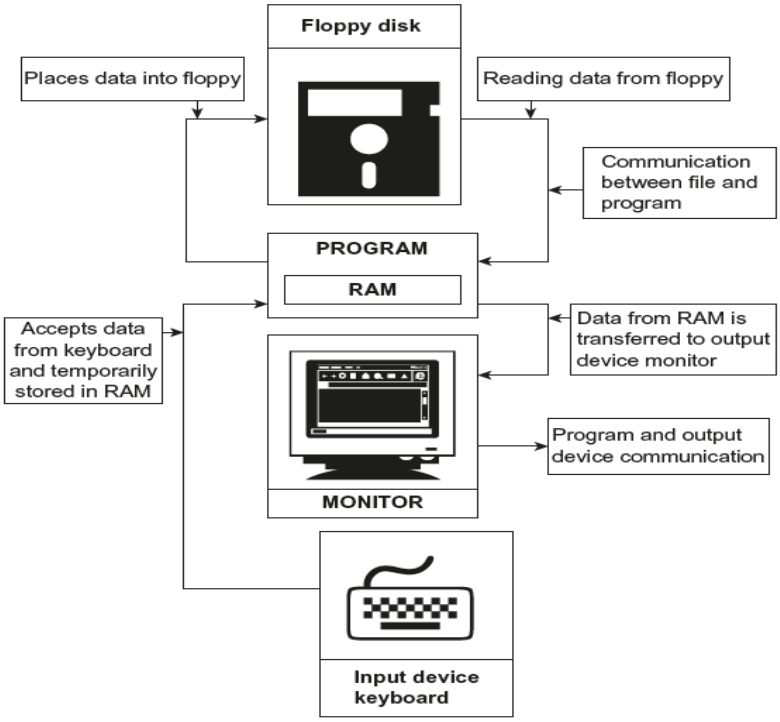


Fig. Communication between program, file, and output device

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.

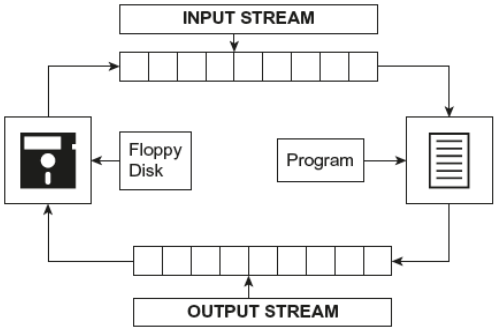
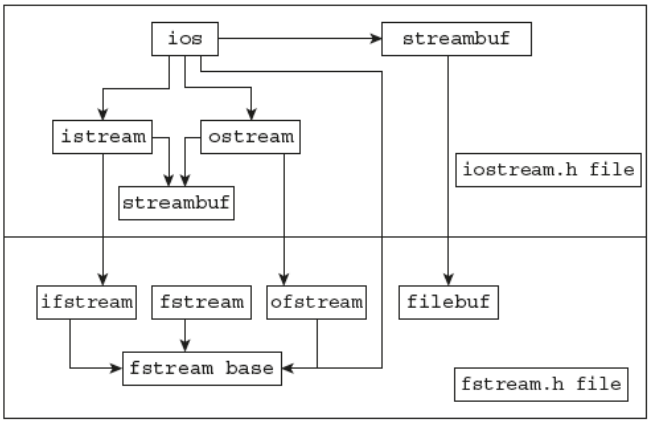


Fig. Input and output streams

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

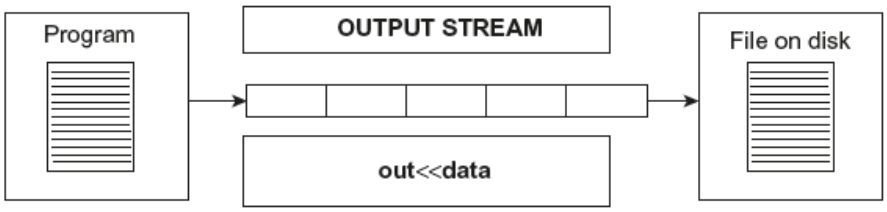
connected with the object in. It is possible to use these file objects in program statements such as stream objects. Consider the following statements:

cout<<"One Two Three";

The above statement displays the given string on the screen.

out<<"One Two Three";

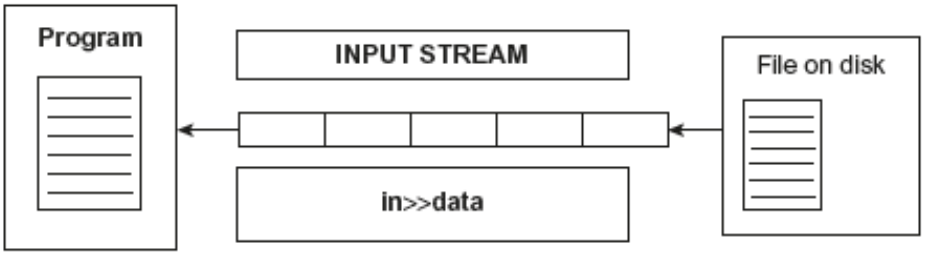
The above statement writes the specified string into the file pointed by the object out as shown in Figure. The insertion operator << has been overloaded appropriately in the ostream class to write data to the appropriate stream.



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 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 is 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 are ios::in (reading only) and for ofstream 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 **iostate** 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.

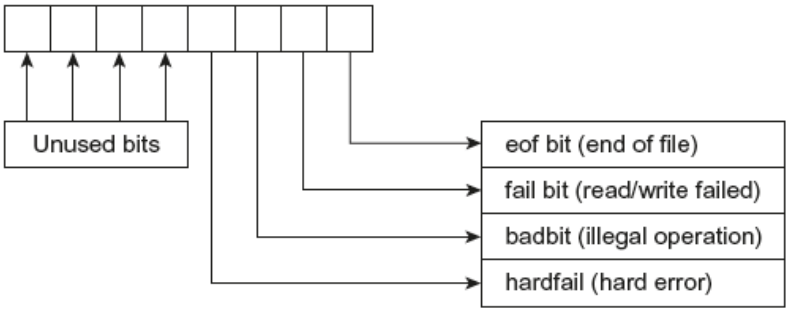


Fig. Status bits

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 .
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,ios::bit are 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.eof()<<endl;
        cout<<"bad():"<<in.bad()<<endl;
        cout<<"good():"<<in.good()<<endl;
        in.close();
        return 0;
}
```

Output:

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

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: 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. Different types of file opening modes are:

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)

/* 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("pvp.txt",ios::out);
        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("pvp.txt",ios::in);
        in>>name;
        in>>age;
        cout<<"\nName:"<<name<<endl;
        cout<<"Age:"<<age;
        in.close();
        return 0;
}
```

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

EMP_CLASS.h

```
#include <iostream>
#include<fstream>
using namespace std;
class emp{
    int empno;
    char name[20];
    float sal;
    public:
        void get();
        void display();
};
void emp::get()
```

```
{
    cout<<"Enter rmpno,name,salary"<<endl;
    cin>>empno>>name>>sal;
}
void emp::display()
{
   cout<<empno<<"\t"<<name<<"\t"<<sal<<endl;
}
```

.cpp File

```
#include"EMP_CLASS.h"
int main() {
        ofstream ofs;
        emp e;
        int i,n;
        ofs.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();
                ofs.write((char *)&e,sizeof(e));
        }
        ofs.close();
        cout<<"writing finished"<<endl;
        cout<<"The data in the file is"<<endl;

        ifstream ifs;
        ifs.open("emp.txt",ios::in);
        while(!ifs.eof())
        {
                ifs.read((char *)&e,sizeof(e));
                e.display();
        }
        ifs.close();
        return 0;
}
```

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

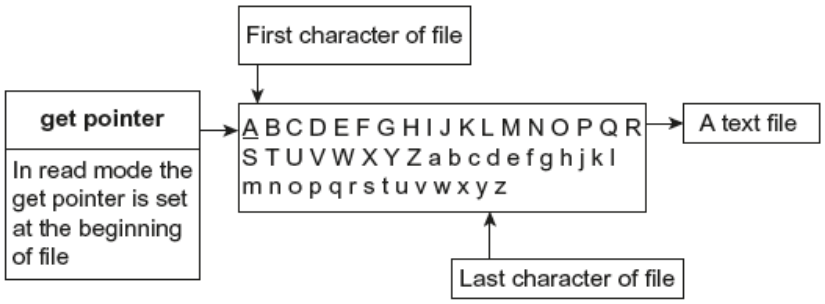
```
 #include<fstream >
using namespace std;
int main()
{

        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;
}
```

File Pointers and Manipulators:

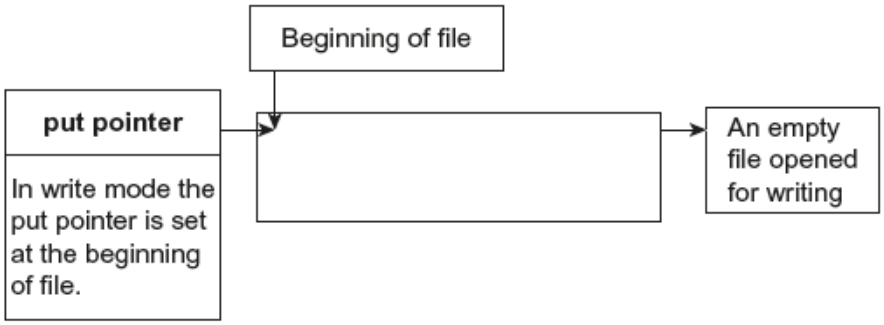
All file objects hold two file pointers that are associated with the file. These two file pointers provide two integer values. These integer values indicate the exact position of the file pointers in the number of bytes in the file. The read or write operations are carried out at the location pointed by these file pointers .One of them is called get pointer (input pointer), and the second one is called put pointer (output pointer).

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:

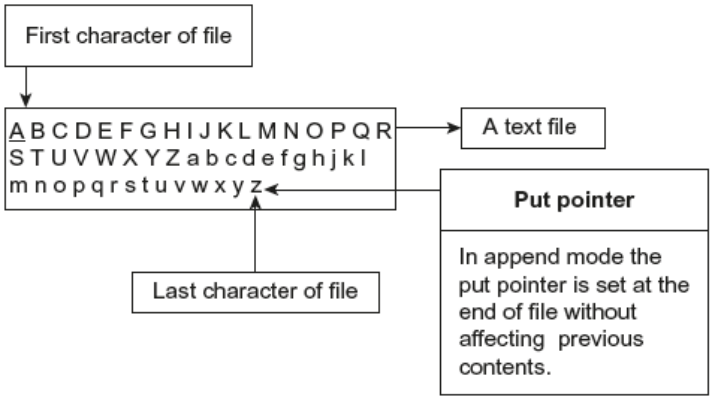
Read mode: When a file is opened in read mode, the get pointer is set at the beginning of the file, as shown in figure. Hence, it is possible to read the file from the first character of the file.



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. Thus, it allows the write operation from the beginning of the file. In case the specified file already exists, its contents will be deleted.



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. Hence, it is possible to write data at the end of the file. In case the specified file not 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.

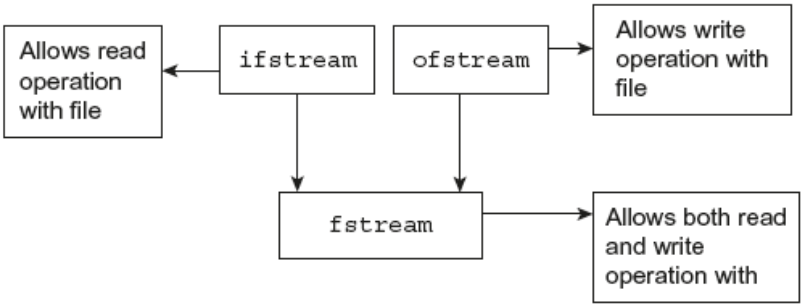


File Pointer Handling Functions:

C++ has four functions for the setting of pointers during file operation. The position of the curser in the file can be changed using these functions. These functions are described in the following table:

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 above table, 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.



/* Write a C++ program to append a file.*/

```
int main()
{
        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;
}
```

/*Write a C++ program to read contents of the file. Display the position of the get pointer.*/

```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
        string data;
        int r;
        ifstream ifs;
        ifs.open("Info.txt");
        cout<<"Contents of the file \n";
        while (ifs)
        {
                r=ifs.tellg();
                ifs>>data;
                cout<<"("<<r <<")"<<data;
        }
        return 0;
}
```

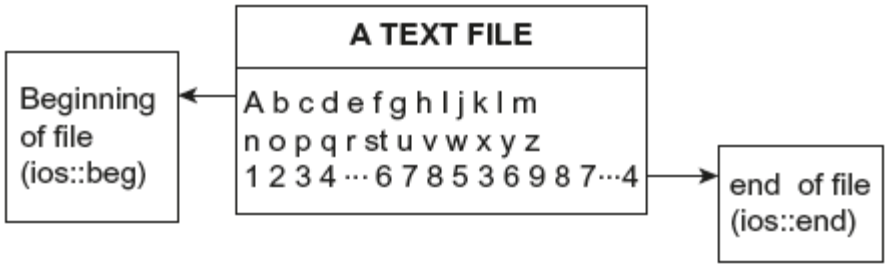
Manipulators with Arguments:

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. 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



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

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

/* 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 <iostream>
#include<fstream>
using namespace std;

int main()
{
   char c;
   char data[20];
   ifstream ifs;
   ofstream ofs;
   ofs.open("text.txt");
   cout<<"Enter Some Data to store in file"<<endl;
   cin.getline(data, 20);
   ofs<<data;
   ofs.close();
   ifs.open("text.txt");
   ifs.seekg(0,ios::end);
   int pos=ifs.tellg();

   for(int i=1;i<=pos;i++)
   {
         ifs.seekg(-i,ios::end);
         ifs>>c;
      cout<<c;
   }
   ifs.close();
   return 0;
}
```

/* Write a C++ program to print the number of records in a file (Employee file) */

```
#include"EMP_CLASS.h"
int main()
{
         ifstream ifs;
         emp e;
         int pos,nor;
         cout<<"No. of Records in the file:";
         ifs.open("emp.dat",ios::binary);
         ifs.seekg(0,ios::end);
         pos=ifs.tellg();
```

```
        nor=pos/sizeof(e);
        cout<<nor<<endl;
        ifs.close();
        return 0;
}
```

/* Write a C++ Program to retrieve nth record from a file (Employee File) */

```
#include"EMP_CLASS.h"
int main()
{
        ifstream fin;
        emp e;
        int pos,nor,rec_no;

        cout<<"No. of Records in the file is:"<<endl;
        fin.open("emp.dat");
        fin.seekg(0,ios::end);
        pos=fin.tellg();
        nor=pos/sizeof(e);
        cout<<nor<<endl;

        cout<<"Enter Record No. To Retrieve\n";
        cin>>rec_no;

        if(rec_no>nor)
        {
                cout<<"Record Does not exist"<<endl;
                exit(1);
        }
        cout<<"The Record At Position: "<<rec_no<<endl;
        pos=(rec_no-1)*sizeof(e);
        fin.seekg(pos,ios::beg);
        fin.read((char *)&e,sizeof(e));
        e.display();
        fin.close();
   return 0;
}
```

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. 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.

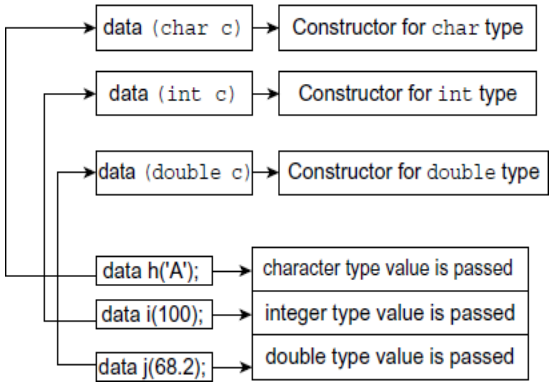


Fig. Working of non-template function

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. */**

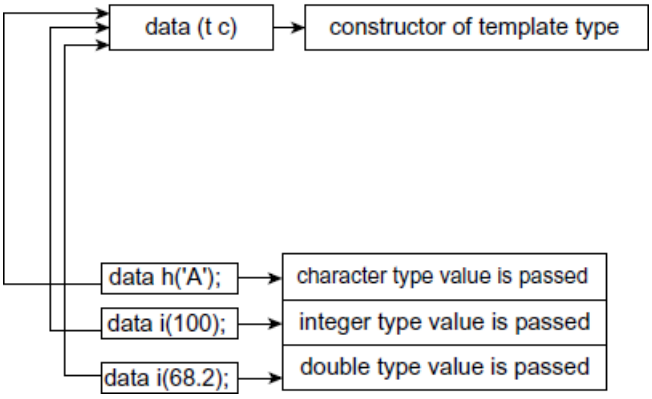
```
#include <iostream>
using namespace std;

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.



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
```

Working of Function Templates: 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
```