# FILE HANDLING IN C++

We all know that RAM is a volatile memory and any data stored in RAM is lost when PC is turned off. All the program we have seen so far have made use of RAM. Any data variable that we define in our program is destroyed when the program execution is over. Also Lhe outputs generated by the program are lost. One solution may be to take printouts of the program and outputs. They may help up to a certain extent but that is not appropriate for the practical purpose. Therefore in most real word applications data is stored in text files which is stored permanently on to the hard disk, floppy, compact disk or in any other persistent storage media. These files can be read back again, and can be modified also.

A data file is a collection of data items stored permanently in persistent storage area. The C++ language provides the facility to create these data files, write data into them, read back data, modify them and many more operations.

The program data or output can be stored in these files and that persists even after program execution is over. The data can be read whenever necessary and can be placed back into the file after modification. The data remains safe provided storage media does not crash or corrupt.

From permanent storage point of view, a file is a region of memory space in the persistent storage media and it can be accessed using built-in library functions and classes available in header file iostream or by the stream calls of the operating systems. High level files are those files which are accessed and manipulated using library functions. For transfer of data they make use of stream.

A stream is a pointer to a buffer of memory which is used for transferring the data. In general stream can be assumed as a sequence of bytes which flow from source to destination .

An I/0 stream may be text stream or binary stream depending upon in which mode you have opened the file. A text stream contains lines of text and the characters in a text stream may be manipulated as per the suitability.

But a binary stream is a sequence of unprocessed data without any modification. The standard 1/0 stream or stream pointers are cin (for reading), cout (for writing), and cerr (for error).

**FILE STREAMS**

In C++ there are three main classes for handling disk files input and output. They are :

(a) ifstream
(b) ofstream
(c) fstream

**The ifstream class** is an istream derived specialized for disk file input. Its constructors automatically create and attach a filebuf buffer object. A file can be created by the constructor method or using the open method of ifstream class. For closing the file close method can be used. ifstream can only be used for reading a file.

**The ofstream class** is an ostream derivative specialized for disk file output. All of its constructors automatically create and associate a filebuf buffer object. A file can be created by the constructor method or using the open method of ofstream class. For closing the file close method can be used. ofstream can only be used for writing to a file only.

**The fstream class** is an iostream derivative specialized for combined disk file input and output. Its constructor automatically create and attach a filebuf buffer object. A file can be created by the constructor or method or using the open method of fstream class. For closing the file close method can be used. The fstream class can be used for reading writing, appending or doing any other operation as well as we will see shortly.

**OPENING AND CLOSING A FILE**

For opening a file any of the above discussed file stream can be used. For opening a file for reading only we can use ifstream class as :

**(a) ifstream rdfile("demo.txt”)**

The above method creates an object of class ifstream type and attaches it to the file "demo.txt”. The file is opened for read only. The object is created by calling the constructor of the class ifstream and passing an argument of char* type which is file name to open. As soon as file is opened for read mode, file pointer is placed at the beginning of the file. We assume that there is no error in opening the file. Error handling will be discussed later on. Assuming the file is opened successfully. We can read from the file as :

```
char str[10];
file>>str;
```

The string read from file (assuming file contains a string "hello”) will be stored in the variable str. The file can be closed later by calling the close method of the class as :

```
file.close( );
```

Calling close method ensures all data stored in the buffer related to the file will be written to the disk and all links will be broken. Always make a practice of closing a file when you are done with the file.

**(b) ifstream rdfile;**

```
rdfile.open ("demo.txt”);
```

This is the second method of opening the file. Here, first an object of ifstream class type is created. We then open the file using the open function of the ifstream class.

On the similar ground we can open a file for writing only the ofstream class as:

(a) ofstream wrfile("demo.txt”);

(b) ofstream wrfile;

(c) wrfile.open("demo.txt”);

Explanation for the above methods is similar to the explanation as given for the ifstream class. The data can be written to the file as :

```
wrfile<<”hello”;
The file can be closed as :
wrfile.close ( );
```

The ifstream class and ofstream class are suitable only when we want to read only from file or write only to the file. In situations when we want to perform reading and writing to the same file. There are two ways :

First is to create two separate objects of ifstream and ofstream class. When writing is done, close the file and open the file for reading using an object of ifstream class.

The second method is to create just one object of class fstream class and use as :

```
fstream file;
file.open("demo.txt”,ios : :out);
//perform writing operation;
file.close( );
file.open("demo.txt”,ios ::in);
//perform reading operation;
```

In the open function of fstream class, second argument is the file opening mode. The mode ios :: in is for reading and ios :: out for writing only. The modes are defined as enumeration constants in the class ios so scope resolution operator is used with it.

Note that there is no ios ::in or ios ::out default mode for fstream objects. You must specify both modes if your fstream object read and write files. We can also have constructor method of fstream for creating and opening a file as :

```
fstream file("num.txt”ios ::in);
char str[10];
file>>str;
cout<<str<<endl;
```

Let’s have some programming examples first then we will see what other modes we can have for handling the files.

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
ofstream fobj("demo.txt");
fobj<<"FILE HANDLING DEMONSTATION "<<endl;
fobj.close( );
return 0;
}
```

Output

Blank screen

The statement ofstream fobj("demo.txt”); creates an object of ostream class type named fobj and passes file name as "demo.txt”. This way of creating file is called constructor notation as we are creating and object of ostream type by calling the constructor which takes an argument of type char* type. After the

execution of the above statement a file stream fobj is created and linked to the file "demo.txt". Now when you want to write something to the file you can write it as shown with the use of <<operator.

Note ofstream class is meant only for writing to the disk files. You cannot read from the file using an object of ofstream. As soon as you open the file using an object of ofstream class type, the file is automatically opened in the write mode. That's why no mode was specified while opening the file. If file "demo.txt" were already existing its contents will be destroyed and file pointer will be placed at the beginning of the file. If file were not already present it will be created.

```
/*READING FROM FILE*/
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{

ifstream fobj;
fobj.open("demo.txt");
char str[30];
fobj.getline(str,30);
cout<<"STRING READ FROM FILE"<<endl;
cout<<str<<endl;
fobj.close();
return 0;
}
```

**Output**

STRING READ FROM FILE

FILE HANDLING DEMO

```
/*WRITING PERSON DATA TO FILE */
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{

ofstream fobj;
fobj.open("demo.txt");
char name[25];
int age;
char sex;
cout<<"ENTER NAME HERE :=";
cin.getline(name,25);
cout<<endl<<"ENTER THE AGE AND SEX :=";
cin>>age>>sex;
fobj<<name<<endl<<age<<endl<<sex<<endl;
fobj.close( );
return 0;
}
```

**OUTPUT :**

ENTER NAME HERE : =VINOD

ENTER THE AGE AND SEX : =24 M

```
/*READING PERSON DATA FROM FILE */
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
ifstream fobj;
fobj.open("demo.txt");
```

```
char name[20];
int age;
char sex;
fobj.getline(name,20);
fobj>>age>>sex;
cout<<"Name :="<<name<<"\nAge :="<<age<<"\nSex :="<<sex<<endl;
fobj.close( );
return 0;
}
```

**Output**

```
Name :=VINOD
Age :=25
Sex :=M
```

**FILE OPENING MODES**

| S.No. | Mode | Meaning |
|---|---|---|
| 1. | ios::in | Opening file in read mode only |
| 2. | ios : :out | Opening file in write mode only |
| 3. | ios : :app | Opening file in append mode |
| 4. | ios : :trunc | File contents deleted if file already exists |
| 5. | ios : :nocreate | In case file does not exist, open function will fail |
| 6. | ios : :noreplace | Opens the file if file already exits |
| 7. | ios : :binary | Open binary file |
| 8. | ios : :ate | File pointer move to end of file on opening the file. |

**1. ios : :trunc**

If the file already exists, its contents are discarded. This mode is implied if ios : :out is specified, and ios : :ate, ios : :app, and ios : :in are not specified.

**2. ios : :noreplace**

If the file already exists, the function fails. This mode automatically opens the files for writing if file does not exist.

**3. ios : :binary**

Opens the file in binary mode (the default in text mode).

**4. ios : :in**

The files is opened for input. The original file (if it exists) will not be truncated.

**5. ios : :ate**

The function performs a seek to the end of file. When the first new byte is written to the file, it is appended to the end, but when subsequent bytes are written, they are written to the current position i.e., any where in the file.

**6. ios : :app**

The function performs a seek to the end of file. When new bytes are written to the file, they are always appended to the end, even if the position is moved with the ostream : :seekp function.

**7. ios : :nocreate**

If the file does not already exist, the function fails.

8. A file can be opened in more than one mode. Number of modes can be combined using binary OR symbol (|)as :

```
file. open("demo.txt”,ios : :in|ios : :out|ios : :binary);
```

```
/*READING AND WRITING THE SAME FILE IN ONE PROGRAM*/

#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
fstream rw;
rw.open("demo1.txt",ios ::out);
char str[50];
cout<<"ENTER A STRING"<<endl;
cin.getline(str,50);
rw<<str<<endl;
rw.close();
rw.open("demo1.txt",ios ::in);
rw.getline(str,50);
cout<<"STRING READ FROM FILE"<<endl;
cout<<str<<endl;
rw.close();
return 0;
}
```

**Output**

```
ENTER A STRING
abcd
STRING READ FROM FILE
abcd
```

**CHECKING END OF FILE**

When file is opened for reading, data from file can be read as long as file is open and till end of file is not encountered. The end of file can be checked in two ways :

(A) In the first method the file stream objects can be written in the while loop as :

```
while (filestr)
{
read from file pointer by file str;
process data;
}
```

As long as we are reading filestr returns nonzero value. As soon as end of the file is encountered, filestr returns zero and while loop terminates.

(B) The second method which is most frequently used is of finding file using eof ( ) function. The function eof returns zero as soon as there is some data in the file i.e., as long as end of file is not reached. As soon as end of the file reached, eof( ) function returns nonzero value. It can be used with while loop as :

```
While(filestr.eof( ) ==0)        or      while(!file.eof( ))
{
read from file pointer by filestr;
process data;
}
```

```
/*EOF FUNCTION AND APPEND MODE */
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
fstream rw;
char str[100];

rw.open("demo.txt",ios ::in);
cout<<"data read from file";
endl(cout);
while(rw.eof()==0)
{
rw.getline(str,100);
cout<<str<<endl;
}
rw.close();
rw.open("demo.txt",ios ::app);
cin.getline(str,100);
rw<<str<<endl;
rw.close();
rw.open("demo.txt",ios ::in);
cout<<"File with append data \n";
while(rw.eof()==0)
{
rw.getline(str,100);
cout<<str<<endl;
}
rw.close();
return 0;
}
```

**Output**

data read from file

VINOD
25
M

```
#include <iostream>
#include <fstream>
int main( )
{
fstream f1,f2;
f1.open(“names.txt,ios::out);
f1<<”Hari”<<endl;
f1<<”Vijay”<<endl;
f1<<”Vivek”<<endl;
f1.close();

f2.open(“sname.txt”,ios::out);
f2<<”Krishnan”<<endl;
f2<<”Nath”<<endl;
f2<<”S”<<endl;
f2.close();
f1.open(“names.txt”,ios::in);
f2.open(“sname.txt”,ios::in);
char str[20];
while(f1.eof()==0)
{
        f1.getline(str,20);
        cout<<str<<” “;
        f2.getline(str,20);
        cout<<str<<endl;
}
f1.close()
f2.close()
```

Output:
Hari

Vijay
Vivek

**RANDOM ACCESS IN FILE**

Random access means reading data randomly from any where in the file. For this purpose we need to set the position of the file pointer first in the file and then read the data. C++ file stream classes provides the following functions for the manipulation of file pointer any where in the file. With the help of these functions we can access data in random fashion.

(a) seekg( )
(b) seekp( )
(c) tellg( )
(d) tellp( )

The function seekg and tellg are used when we are reading from file i.e., when used with the get pointer that's why the suffix g. The function tellp and seekp are used when writing to the file i.e., when put pointer is used that's why the suffix p. The seek (p/g) function moves pointer to the specified position and tell (p/g) gives the position of the current pointer. The most frequently used function is seekg whose syntax is as :

istream & seekg (streamoff off, ios : :seek_dir dir);

The **off** is the new offset value and **streamoff** is a typedef equivalent to long. offset positive means move forwards, −ve means move backwards and 0 means stay at the current position. First byte in the file is at position 0. The **dir** is the seek diretion. Must be done of the following enumerators :

→ ios : :beg Seek from the beginning of the stream.
→ ios : :cur Seek from the current position in the stream.
→ ios : : end Seek from the end of the stream.

ifstream file;
file.seekg(2,ios : :beg)-> Third byte from the beginning.
file.seekg(6,ios : :cur)-> Forward by 6 bytes from current pos.
file.seekg(-5,ios : :cur)->Backwardby 5 bytes from current pos.

file.seekg(-4,ios : :end)-> Backward by 4 bytes from end.

file.seekg(0,ios : :cur)-> stay at the current position.

Similar, to seekg you can use seekp for moving pointer within a file which is opened for writing. The function tellg and tellp can be used for finding number of bytes into the file by opening the file and moving pointer to the end of the file. They can be used as :

```
ofstream file("num.txt”, ios : :ate);
cout<<file.tellp( )<<endl;
```

When file is opened the pointer will be at the end of the file and tellp will give number of bytes in the file.

Similarly, we can use tellg as :

```
ifstream file("num.txt”, ios : :in);
file.seekg(0, ios : :end);
cout<<file.tellg( )<<endl;
```

When file is opened for reading, the file pointer will be in the beginning of the file so we have moved pointer to the end of the file using seekg.

**Program**

Write a program to read following sentence and to reverse it.

Love is blind

I Love Programming

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
fstream file;
int count, x=1;
char ch;
```

```
file.open(“num.txt”,ios : :in);
cout<<“file contents”;
endl(cout);
while(file.eof( )==0)
{
file.get(ch);
cout.put(ch);
}
endl(cout);
cout<<“Reversing contents”;
endl(cout);
file.clear( );
file.seekg(0,ios : :end);
count=file.tellg( );
while(x<=count)
{
file.seekg(−x,ios : :end);
file.get(ch);
cout.put(ch);
x++;
}
cout<<endl;
file.close( );
return 0;
}
```

UTPUT :

file contents

Love is blind

I Love programming

Reversing contents

gnimmargorp evoL I

dnilb si evoL

**Find the length of file**

fstream file

file.open()

int count;

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

count=file.tellg()

```
#include<iostream>
#include<fstream>

using namespace std;
int main()
{
        fstream file;
        int count,x=1;
        char ch;

file.open("test.txt",ios::in);
endl(cout);

while(file.eof()==0)
{
        file.get(ch);
        cout.put(ch);
}
endl(cout);

//Counting no of bytes
file.seekg(0,ios::end);
count=file.tellg();

//Reverse the file contents
while(x<=count)
{
        file.seekg(-x,ios::end);
```

```
        file.get(ch);
        cout.put(ch);
        x++;
}
cout<<endl;
file.close();
}
```

**Program**

Assume we have a simple file with the contents shown below:

There are wonderful things to do in life.

We want to change the contents of the file so each word is on a line by itself.

- ➔ Note that we read a character at a time (sequential access).
- ➔ We then check the value of the character. If it is a space, we change it to carriage return character ('\r')
- ➔ After reading a character and finding that it is a space, we must change it. However, the marker has already advanced to the beginning of the next word. We must move the marker back before putting a carriage return in place of the space character.
- ➔ We do this by using the seekp function to move the marker back one position.

** A program to put each word in the file on a new line*

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
fstream fstr;
fstr.open ("file3" , ios :: in | ios :: out);
char ch;
while (fstr.get(ch))
{
```

```
if (isspace (ch))
{
fstr.seekp (−1 , ios :: cur);
fstr.put ('\r');
}
}
fstr.close ();
return 0;
}
```

**Output:**

There
are
wonderful
things
to
do
in
life.

**Exercise:**

To count number of lines and characters

**Program:**

we can find the size of a file in bytes if we open the file using the ate (at the end) opening mode, which puts the marker after the last character in the file. We can then use the tellg member function to tell us the index of the character after the last character,

**This is the file whose size we want to find.**

**Output:** File size: 44

```
* A program to find the length of the file

#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
// Instantiation of stream and connection to the file
ifstream ifstr;
ifstr.open ("file4" , ios :: in | ios :: ate);
// Finding the marker value after the last character
cout << "File size: " << ifstr.tellg ();
// Closing the file
ifstr.close ();
return 0;
}
```

**COMMAND LINE ARGUMENTS**

In all the programs we have used so far we have written main( ) without arguments but in reality main ( ) does take arguments and also have a return type. The prototype of main( ) in C++ is given as :

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

Number of parameters are passed are collected into the parameter argc which stands for argument count. Value of each argument is stored in string array argv which stands for argument value. All arguments are of string type whether they are int, float or even a character. By default return type of any function if not specified int is assumed that's why int before main( ) is optional. For better understanding point of view see the following examples.

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

int main(int argc, char* argv[])
{
        int i;
        for(i=1;i<argc;i++)
                cout<<”Value= “<<i<<argv[i]<<endl;
        return 0;
}
```

**Output**

```
Value=1 Hello
Value=2 Welcome
Value=3 To C++
```

**Programs**

1. Sum of command line argument

atoi ->stdlib.h To convert string to integer

```
#include<iostream>
#include<stdlib.h>
using namespace std;

int main(int argc, char **argv)
{
        int i, sum=0;
        for(i=0;i<argc;i++)
                sum=sum+atoi(argv[i])
        cout<<”Sum is : “<<sum<<endl;
return 0;
}
```

**Output**:

./a 1 2 3 4 5 6 7 8 9
Sum is 45

**Write a program to find Max of command line arguments, float data**

```
#include<iostream>
#include<stdlib.h>
using namespace std;

int main(int argc, char **argv)
{
        int i;
        float max,t;
        max=atof(argv[1]);
        for(i=1;i<argc;i++)
        {       t=atof(argv[i]);
                if(max<t)
                        max=t;
        }
        cout<<"Max is : "<<max<<endl;
return 0;
}
```

./a 2.3 5.8 7.9 **85.6** 45.9
Max is 85.6

**Write a Program for Sorting command line arguments**

```
#include<iostream>
#include<string>

using namespace std;

int main(int argc, char **argv)
{
        char *temp;
        int i,j;
        for(i=1;i<argc,i++)
        {
                for(j=i+1;j<argc;j++)
                        if(strcmp(argv[i],argv[j]>0)
                        {
                                temp=argv[i];
                                argv[i]=argv[j];
                                argv[j]=temp;
                        }
        }
        for(i=1;i<argc;i++)
                cout<<argv[i]<<endl;
        return 0;
}
```

```
./a ONE TWO THREE FOUR
FOUR
ONE
THREE
TWO
```

**Program for Copying of one file into another with file names supplied as Command line arguments**

```
#include<iostream>
#include<fstream>
#include<stdlib>

using namespace std;

int main(int argc, char **argv)
{
        fstream f1,f2;
        char ch;
        cout<<argc<<endl;

        if(argc!=3)
        {
                cout<<”Input <<filename<source file<destination file”;
                exit(0);
        }

        f1.open(argv[1],ios::in);
        f2.open(argv[2],ios::out);

        if(f1.fail())
        {
                cout<<”File opening error”;
                exit(0);
        }

        while(f1.eof()==0)
        {
                f1.get(ch);
```

```
            f2.put(ch);
        }
        f1.close();
        f2.close();
        return 0;
}
```

**WORKING WITH BINARY MODE**

Numeric data including integer and floating point as well as character data, all are treated in terms of character data i.e., string "file" will take 4 bytes in memory, integer number 1245 will take 4 bytes in memory and even 1.23 will take 4 bytes in memory. But as these data stored in disk file they are stored in binary form and in this they are stored as per their type i.e., integer takes 2 bytes, float takes 4 bytes and character takes 1 bytes. So, if reading and writing is done when treating file as text file and it contains lots of numerical data it will require large amount of disk space. For that we can have two functions provided by C++ stream classes read and write which reads and write data in terms of binary. For opening file in binary mode we have the mode ios ::binary. The prototype of both the function is given as :

```
        Ostream & write(const char * pch, int nCount);
        Istream & read(char * pch, int nCount);
```

The first argument in the write is the address of the character array and second is the number of characters to be written. Similar arguments apply to fread with the difference that it reads instead of writing.

```
#include<iostream>
#include<fstream>
#include<string>

using namespace std;

class person
{
        char name[10];
        int age;
public:
        void input(char n[], int a)
        {
                strcpy(name,n);
                age=a;
        }

        void show()
        {
                cout<<"Name= "<<name<<endl;
                cout<<"Age= "<<age<<endl;
        }
};

int main()
{
        fstream fp;
        fp.open("test.txt", ios::in | ios::out);
        person p;
        p.input("Vinod", 22);
        fp.write((char*) &p,sizeof(p));
        fp.read((char*) &p,sizeof(p));
        p.show();
        fp.close();
        return 0;
```

}

Output:
Name= Vinod
Age = 22

**Exercise:** Write a Student Database Management program
Add a record
View Record
Search Record
Delete a Record
Modify a Record
Count Record
Exit
student.txt -> temp.txt remove
Name, rollno, class
loc=(n-1)*sizeof(obj)

**ERROR HANDLING**

When dealing with the file errors might occurs such as :

- # File does not exist
- # Reading from file which is opened for writing only.
- # Path is not valid.
- # File already exist etc.

To cope up with all these error we check whether file is opened successfully or what type of error has beem generated. Some of the error handling functions with their descripton is given below :

**1. The good Function**

SYNTAX : int good( ) const;

Returns nonzero values if all error bits are clear. Note that the good member function is not simply the inverse of bad function.

**2. The bad Function**

SYNTAX : int bad( ) const;

Returns a nonzerovalue to indicates a serious I/O error. This is the same as setting the badbit error state. Do not continue I/O operations on the stream is this situtation.

**3. The fail Function**

SYNTAX : int fail ( ) const;

Returns a nonzero value if any I/O error (not end of file) has occurred. This condition corresponds to either the badbit or failbit error flag being set. If a call to bad returns 0, you can assume that the error condition is nonfatal and that you can probably continue processing after you clear the flags.

**4. The clear Function**

SYNTAX : void clear (int nState = 0);

The parameter nState, if 0, then all error bits are cleared, otherwise bits are set according to the following masks(ios enemerators) that can be combined using the bitwise OR (|) operator.

The nState parameter must have one of the following values :

-> ios : :goobit No error condition (no bits set).

-> ios : :eofbit End of file reached.

-> ios : :failbit A possibly recoverable formatting or coversion error.

-> ios : :badbit A server I/O error.

The function sets or clear the error-state flags. The rdstate function can be used to read the current error state.

**5. The eof( ) Function**

SYNTAX : int eof( ) const;

Returns a nonzero value if end of file has been reached. This is the same as setting the eofbit error flag.

**6. The rdstate Function**

SYNTAX :int rdstate ( ) const;

Returns the current error state as specified by the following masks (ios enumerators).

-> ios : :goobit No error condition (no bits set).

-> ios : :eofbit End of file reached.

-> ios : :failbit A possibly recoverable formatting or coversion error.

-> ios : :badbit A server I/O error or unknown state.

This is shown in the figure given below. The last 4 bits are unused.The returned value can be tested against a mask with the AND (&) operator, but we do not have to as we can have functions which tells us which bit is set or reset.


Error state specified by the masks

**Program**

Error handling with files

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

int main()
{
	ofstream file("test.txt",ios::noreplace);
	if(!file)
	{
		cout<<"File opening error\n";
		endl(cout);
	}
//rdstate
cout<<"rdstate="<<file.rdstate();
```

```
//bad
cout<<”bad=”<<file.bad();

//fail
cout<<”fail=”<<file.fail();

//good
cout<<”good=”<<file.good();

//eof
cout<<”eof=”<<file.eof();

return 0;

}
```