UNIT-IV

C++ Files and Streams

File represents storage medium for storing data or information. Streams refer to sequence of bytes. In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data. So we use the term File Streams/File handling. We use the header file `<fstream>`

- **ofstream:** It represents output Stream and this is used for writing in files.
- **ifstream:** It represents input Stream and this is used for reading from files.
- **fstream:** It represents both output Stream and input Stream. So it can read from files and write to files.

Operations in File Handling:

- Creating a file: `open()`
- Reading data: `read()`
- Writing new data: `write()`
- Closing a file: `close()`

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.

3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.
6	**Ios::binary** causes a file to be opened in binary mode

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

Creating/Opening a File

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating. Syntax for file creation: `FilePointer.open("Path",ios::mode);`

- Example of file opened for writing: `st.open("E:\studytonight.txt",ios::out);`
- Example of file opened for reading: `st.open("E:\studytonight.txt",ios::in);`
- Example of file opened for appending: `st.open("E:\studytonight.txt",ios::app);`
- Example of file opened for truncating: `st.open("E:\studytonight.txt",ios::trunc);`

However, you will often see open( ) called as shown, because the mode parameter provides default values for each type of stream. As their prototypes show, for ifstream, mode defaults to ios::in; for ofstream, it is ios::out | ios::trunc; and for fstream, it is ios::in | ios::out. Therefore, the preceding statement will usually look like this:

```
out.open("test"); // defaults to output and normal file
```

note: Depending on your compiler, the mode parameter for fstream::open( ) may not default to in | out. Therefore, you might need to specify this explicitly.

If open( ) fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded. You can do so by using a statement like this:

```
if(!mystream)
{
cout << "Cannot open file.\n"; // handle error
}
```

Although it is entirely proper to open a file by using the open( ) function, most of the time you will not do so because the ifstream, ofstream, and fstream classes have constructors that automatically open the file. The constructors have the same parameters and defaults as the open( ) function. Therefore, you will most commonly see a file opened as shown here:

```
ifstream mystream("myfile"); // open file for input
```

As stated, if for some reason the file cannot be opened, the value of the associated stream variable will evaluate to false. Therefore, whether you use a constructor to open the file or an explicit call to open( ), you will want to confirm that the file has actually been opened by testing the value of the stream.

You can also check to see if you have successfully opened a file by using the is_open( ) function, which is a member of fstream, ifstream, and ofstream. It has this prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if mystream is currently open:

```
if(!mystream.is_open())
{ cout << "File is not open.\n";
// ...
```

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

using namespace std;

int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created";
        st.close(); // Step 4: Closing file
```

```
        }
        getch();
        return 0;
    }
```

Writing to a File

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

using namespace std;

int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created";
        st<<"Hello";    // Step 4: Writing to file
        st.close(); // Step 5: Closing file
    }
    getch();
    return 0;
}
```

Here we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after **"FilePointer <<"** will be passed to output file.

Reading from a File

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

using namespace std;

int main()
{
    fstream st; // step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::in);   // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"No such file";
    }
    else
    {
        char ch;
        while (!st.eof())
        {
```

```
            st >>ch;  // Step 4: Reading from file
            cout << ch;   // Message Read from file
        }
        st.close(); // Step 5: Closing file
    }
    getch();
    return 0;
}
```

Here we are reading input from a file. So, we use ios::in. As given in the program, information from the output file is obtained with the help of following syntax **"FilePointer >>variable"**.

Reading and Writing Text Files It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using cin and cout, substitute a stream that is linked to a file. For example, this program creates a short inventory file that contains each item's name and its cost:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream out("INVNTRY"); // output, normal file
if(!out) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
out << "Radios " << 39.95 << endl;
out << "Toasters " << 19.95 << endl;
out << "Mixers " << 24.80 << endl;
out.close();
return 0;
}
```

The following program reads the inventory file created by the previous program and displays its contents on the screen:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream in("INVNTRY"); // input
if(!in) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
char item[20];
float cost;
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
```

```
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in.close();
return 0;
}
```

In a way, reading and writing files by using >> and << is like using the C-based functions fprintf( ) and fscanf( ). All information is stored in the file in the same format as it would be displayed on the screen.

write(),read() function examples:

write() example:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
class stu
{
int id;
char name[20];
public:
        void getstu()
        {
        cout<<"enter student id, name:";
        cin>>id>>name;
        }
};

void main()
{
stu s; //object
//ofstream file("stu.dat"); //using parameterized constructor
 ofstream file;
file.open("stu.dat"); //using open member function
char op;
do
{
s.getstu();
file.write((char *)&s,sizeof(s));
cout<<"one row created\n";
cout<<"do want to insert another record[y/n]:";
cin>>op;
}while(op=='y'||op=='Y');
file.close();
```

```
getch();
}
```

read() example:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
class stu
{
int id;
char name[20];
public:
        void putstu()
        {
        cout<<id<<"\t"<<name<<endl;
        }
};

void main()
{
stu s;
ifstream file("stu.dat",ios::in);
clrscr();
file.read((char *)&s,sizeof(s));
cout<<"id \t name\n";
cout<<"_____\n";
while(!file.eof())
{
s.putstu();
file.read(char *)&s,sizeof(s));
}
file.close();
getch();
}
```

Another example:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
class stu
{
int id;
char name[20];
public:
        void enq();
};
```

```
void stu:: enq()
{
stu s;
ifstream file("stu.dat");
int idno,found=0;
cout<<"enter student id:";
cin>>idno;
file.read((char *)&s,sizeof(s));
while(!file.eof())
{
if(s.id==idno)
{
cout<<"id\tname\n";
cout<<"_____\n";
cout<<s.id<<"\t"<<s.name<<endl;
found=1;
break;
}
file.read((char *)&s,sizeof(s));
}// while close
file.close();
if(found==0)
cout<<"no data found";
}//enq close

void main()
{
stu s;
s.enq();
getch();
}
```

Following is another example of disk I/O. This program reads strings entered at the keyboard and writes them to disk. The program stops when the user enters an exclamation point. To use the program, specify the name of the output file on the
command line.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: output <filename>\n";
return 1;
```

```
}
ofstream out(argv[1]); // output, normal file
if(!out) {
cout << "Cannot open output file.\n";
return 1;
}
char str[80];
cout << "Write strings to disk. Enter ! to stop.\n";
do {
cout << ": ";
cin >> str;
out << str << endl;
} while (*str != '!');
out.close();
return 0;
}
```

When reading text files using the >> operator, keep in mind that certain character translations will occur. For example, white-space characters are omitted. If you want to prevent any character translations, you must open a file for binary access and use the functions discussed in the next section.

Close a File

It is done by `FilePointer.close()`.

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

using namespace std;

int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new file
    st.close(); // Step 4: Closing file
    getch();
    return 0;
}
```

Special operations in a File

There are few important functions to be used with file streams like:

- `tellp()` - It tells the current position of the put pointer.

Syntax: filepointer.tellp()

- `tellg()` - It tells the current position of the get pointer.

  Syntax: filepointer.tellg()

- `seekp()` - It moves the put pointer to mentioned location.

  Syntax: filepointer.seekp(no of bytes,reference mode)

- `seekg()` - It moves get pointer(input) to a specified location.

  Syntax: filepointer.seekg((no of bytes,reference point)

- `put()` - It writes a single character to file.
- `get()` - It reads a single character from file.

Note: *For seekp and seekg three reference points are passed:*
ios::beg *- beginning of the file*
ios::cur *- current position in the file*
ios::end *- end of the file*

get() function example:

```
#include <iostream.h>

#include <fstream.h>

#include<conio.h>

void main()

{

ifstream in;

char ch;

in.open("demo.cpp");

while(in)

{

in.get(ch);

if(in)

cout<<ch;
```

}

getch();

}

C++ getline()

The cin is an object which is used to take input from the user but does not allow to take the input in multiple lines. To accept the multiple lines, we use the getline() function. It is a pre-defined function defined in a **<string.h>** header file used to accept a line or a string from the input stream until the delimiting character is encountered.

```
#include <iostream>
#include<string.h>
using namespace std;
int main()
{
string name; // variable declaration.
std::cout << "Enter your name :" << std::endl;
getline(cin,name); // implementing a getline() function
cout<<"\nHello "<<name;
return 0;
}
```

Output

```
Enter your name :
John Miller
Hello John Miller
```

Another example:

```
#include<iostream>

#include<fstream.h>

using namespace std;

int main()
{
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt",ios::in);

// Use a while loop together with the getline() function to read the file line by
```

```
line
while (getline (MyReadFile, myText))
{
  // Output the text from the file
  cout << myText;
}

// Close the file
MyReadFile.close();
}
```

Below is a program to show importance of `tellp`, `tellg`, `seekp` and `seekg`:

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

using namespace std;

int main()
{
    fstream st; // Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Creating new file
    if(!st) // Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created"<<endl;
        st<<"Hello Friends"; //Writing to file

        // Checking the file pointer position
        cout<<"File Pointer Position is "<<st.tellp()<<endl;

        st.seekp(-1, ios::cur); // Go one position back from current position

        //Checking the file pointer position
        cout<<"As per tellp File Pointer Position is "<<st.tellp()<<endl;
```

```
        st.close(); // closing file

    }

    st.open("E:\studytonight.txt",ios::in);   // Opening file in read mode

    if(!st) //Checking whether file exist

    {

        cout<<"No such file";

    }

    else

    {

        char ch;

        st.seekg(5, ios::beg);  // Go to position 5 from begning.

        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file
pointer position

        cout<<endl;

        st.seekg(1, ios::cur); //Go to position 1 from beginning.

        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file
pointer position

        st.close(); //Closing file

    }

    getch();

    return 0;

}
```

```
OUTPUT:

New file created

File Pointer Position is 13

As per tellp File Pointer Position is 12

As per tellg File Pointer Position is 5

As per tellg File Pointer Position is 6
```

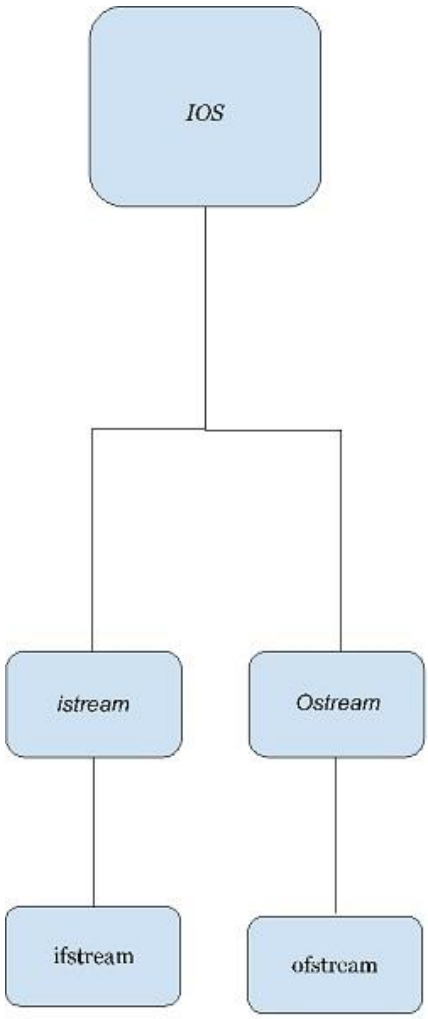
Stream classes hierarchy

In C++ stream refers to the stream of characters that are transferred between the program thread and i/o.

Stream classes in C++ are used to input and output operations on files and io devices. These classes have specific features and to handle input and output of the program.

The **iostream.h** library holds all the stream classes in the C++ programming language.

Let's see the hierarchy and learn about them,



Now, let's learn about the classes of the *iostream* library.

ios class − This class is the base class for all stream classes. The streams can be input or output streams. This class defines members that are independent of how the templates of the class are defined.

istream Class − The istream class handles the input stream in c++ programming language. These input stream objects are used to read and interpret the input as a sequence of characters. The cin handles the input.

ostream class − The ostream class handles the output stream in c++ programming language. These output stream objects are used to write data as a sequence of characters on the screen. cout and puts handle the out streams in c++ programming language.

Example

OUT STREAM

COUT

```
#include <iostream>

using namespace std;

int main(){
```

```
   cout<<"This output is printed on screen";
}
```

Output

```
This output is printed on screen
```

PUTS

```
#include <iostream>
using namespace std;
int main(){
   puts("This output is printed using puts");
}
```

Output

```
This output is printed using puts
```

IN STREAM

CIN

```
#include <iostream>
using namespace std;
int main(){
   int no;
   cout<<"Enter a number ";
   cin>>no;
   cout<<"Number entered using cin is "<
```

Output

```
Enter a number 3453
Number entered using cin is 3453
```

gets

```
#include <iostream>
using namespace std;
int main(){
   char ch[10];
```

```
   puts("Enter a character array");

   gets(ch);

   puts("The character array entered using gets is : ");

   puts(ch);

}
```

Output

```
Enter a character array
thdgf
The character array entered using gets is :
thdgf
```

Error handling during file operations

Sometimes during file operations, errors may also creep in. For example, a file being opened for reading might not exist. Or a file name used for a new file may already exist. Or an attempt could be made to read past the end-of-file. Or such as invalid operation may be performed. There might not be enough space in the disk for storing data.

To check for such errors and to ensure smooth processing, C++ file streams inherit 'stream-state' members from the ios class that store the information on the status of a file that is being currently used. The current state of the I/O system is held in an integer, in which the following flags are encoded :

Name	Meaning
eofbit	1 when end-of-file is encountered, 0 otherwise.
failbit	1 when a non-fatal I/O error has occurred, 0 otherwise
badbit	1 when a fatal I/O error has occurred, 0 otherwise
goodbit	0 value

C++ Error Handling Functions

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

Following table lists these error handling functions and their meaning :

Function	Meaning
int bad()	Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
int eof()	Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
int fail()	Returns non-zero (true) when an input or output operation has failed.
int good()	Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.
clear()	Resets the error state so that further operations can be attempted.

The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set. The fail() function returns true if failbit is set; the good() returns true there are no errors. Otherwise, they return false.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures. For example :

```
:
ifstream fin;
fin.open("master", ios::in);
while(!fin.fail())
{
        :         // process the file
}
if(fin.eof())
{
        :          // terminate the program
}
else if(fin.bad())
{
        :          // report fatal error
}
else
{
        fin.clear();      // clear error-state flags
        :
}
:
```

C++ Error Handling Example

Here is an example program, illustrating error handling during file operations in a C++ program:

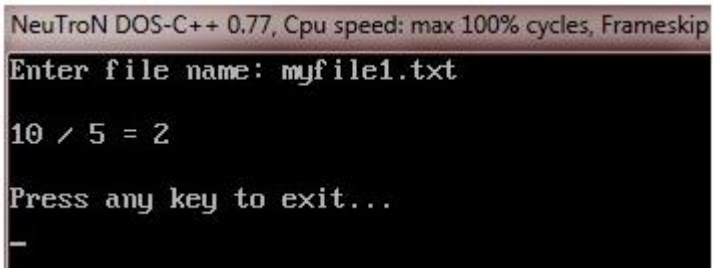
```
/* C++ Error Handling During File Operations
 * This program demonstrates the concept of
 * handling the errors during file operations
 * in a C++ program */

#include<iostream.h>
#include<fstream.h>
#include<process.h>
#include<conio.h>
void main()
{
        clrscr();
        char fname[20];
        cout<<"Enter file name: ";
        cin.getline(fname, 20);
        ifstream fin(fname, ios::in);
        if(!fin)
        {
                cout<<"Error in opening the file\n";
                cout<<"Press a key to exit...\n";
                getch();
                exit(1);
        }
        int val1, val2;
        int res=0;
        char op;
        fin>>val1>>val2>>op;
        switch(op)
        {
                case '+':
                        res = val1 + val2;
                        cout<<"\n"<<val1<<" + "<<val2<<" = "<<res;
                        break;
                case '-':
                        res = val1 - val2;
                        cout<<"\n"<<val1<<" - "<<val2<<" = "<<res;
                        break;
                case '*':
                        res = val1 * val2;
                        cout<<"\n"<<val1<<" * "<<val2<<" = "<<res;
                        break;
                case '/':
                        if(val2==0)
                        {
                                cout<<"\nDivide by Zero Error..!!\n";
                                cout<<"\nPress any key to exit...\n";
                                getch();
                                exit(2);
                        }
                        res = val1 / val2;
                        cout<<"\n"<<val1<<" / "<<val2<<" = "<<res;
                        break;

        }

        fin.close();
```

```
        cout<<"\n\nPress any key to exit...\n";
        getch();
}
```

Let's suppose we have four files with the following names and data, shown in this table:

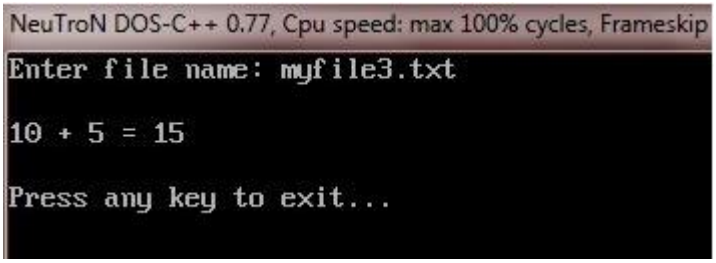
File Name	Data
myfile1.txt	10 5 /
myfile2.txt	10 0 /
myfile3.txt	10 5 +
myfile4.txt	10 0 +

Now we are going to show the sample run of the above C++ program, on processing the files listed in the above table. Here are the four sample runs of the above C++ program, processing all the four files listed in the above table. Here is the sample output for the first file.



This output is for the second file

This is for the third file



This output produced, if the fourth file is processed.