12. Basics of File Handling

12.1 Opening, reading, and writing of files

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:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
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 the **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.

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

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 it 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 closes 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 & Write Example:

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

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

int main ()
{

   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;
   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();
   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output:

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers:

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream,**ios::cur** for positioning relative to the current position in a stream or **ios::end**for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

12.2 Error handling during files operation

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

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.

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 :

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:

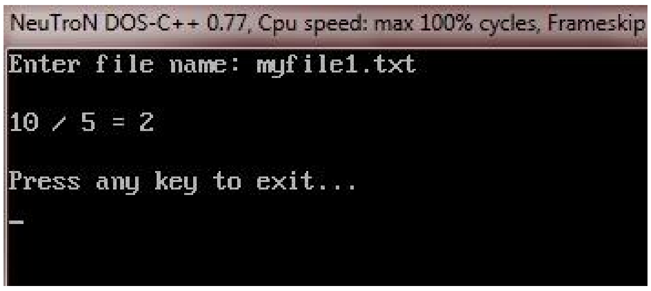
```
#include<iostream.h>
#include<fstream.h>
#include<process.h>
#include<conio.h>
void main()
{
        clrscr();
        char fname[20];
        cout<<"Enter file name: ";
        cin.getline(fname, 20);
        ifstream fin(fname, ios::in);
        if(!fin)
        {
                cout<<"Error in 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 +

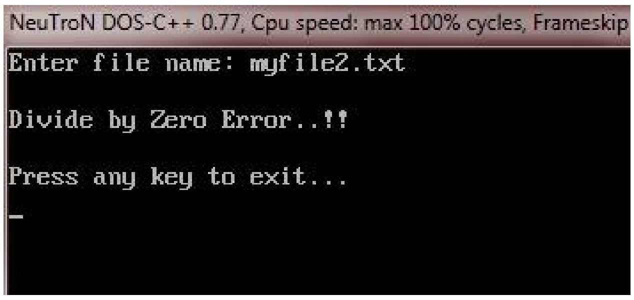
Now we are going to show the sample run of the above C++ program, on processing the files listed in the above table. Here are the four sample runs of the above C++ program, processing all the four files listed in the above table. Here is the sample output for the first file.



NeuTroN DOS-C++ 0.77, Cpu speed: max 100% cycles, Frameskip

```
Enter file name: myfile1.txt

10 / 5 = 2

Press any key to exit...
_
```

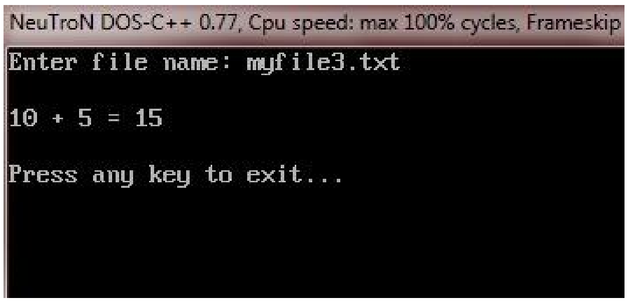
This output is for the second file



NeuTroN DOS-C++ 0.77, Cpu speed: max 100% cycles, Frameskip

```
Enter file name: myfile2.txt

Divide by Zero Error..!!

Press any key to exit...
_
```

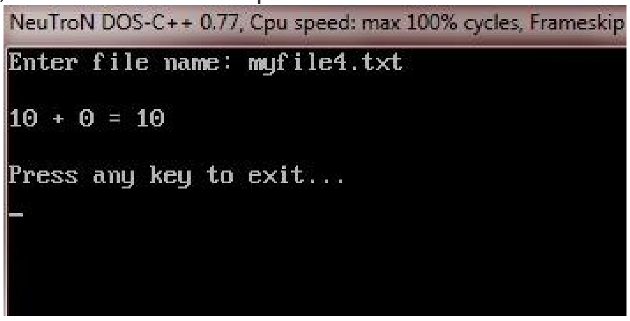
This is for the third file



NeuTroN DOS-C++ 0.77, Cpu speed: max 100% cycles, Frameskip

```
Enter file name: myfile3.txt

10 + 5 = 15

Press any key to exit...
```

This output produced, if the fourth file is processed.



NeuTroN DOS-C++ 0.77, Cpu speed: max 100% cycles, Frameskip

```
Enter file name: myfile4.txt

10 + 0 = 10

Press any key to exit...
_
```