**Error handling in file 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 |

**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";

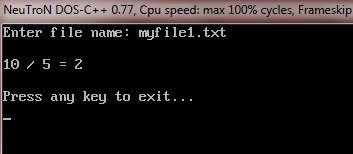
getch();

}

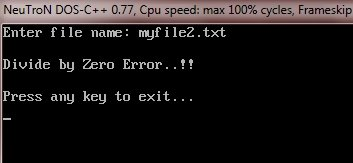
Let's suppose we have four files with the following names and data, shown in this table:

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

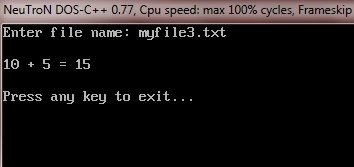
Now we are going to show the sample run of the above C++ program, on processing the files listed in the above table. Here are the four sample runs of the above C++ program, processing all the four files listed in the above table. Here is the sample output for the first file



output for the second file



output for the third file



output for the fourth file

