OOP
Lecture 15

2/12/2021

File Handling in C++ :

File Handling is the storing of data in a file using a program. In C++ programming language, the programs store results, and other data of the program to a file using file handling operations in C++. Also, we can extract/fetch data from a file to work with it in the program. There are 2 major advantages of storing data in files:

- Permanent Storage of Data
- Large Input / Output can be handled easily

These are the basic operations which we can do with files in C++ :

- Create a file
- Open a file
- Read from a file
- Write to a file
- Close a file

Classes for file Handling :

In C++, files are mainly dealt with by using three classes **fstream, ifstream, ofstream** available in **<fstream>** header file.

ofstream: Stream class to **write** data in files
ifstream: Stream class to **read** data from files
fstream: Stream class to **both read and write data** from/to files.

Some of the basic functions for file handling in C++ are :

- open( )
- close( )
- fail( )
- bad( )
- clear( )
- seekg( )
- seekp( )
- tellg( )
- tellp( )

Opening Files in C++ :

We can open any file by using 2 methods,

1. We can open a file by writing

   class_name object_name("file_name", mode);

Class_name here can be any of the three above mentioned classes, depending on our requirement we can use ofstream, ifstream or fstream.

object_name is like the name of the variable, we can give any name here and call our file with this name.

In parentheses, we pass two arguments, which will be passed to the constructor of the corresponding class. The first parameter is the complete name of the file with extension if the file is placed in the same directory in which our code is stored (called pwd : present working directory). And if the file is placed in any other directory then we need to pass the complete path + name of the file in parameter, remember that the path and filename should be enclosed in the double quotes. The second parameter is optional. It specifies the mode of opening the file. We will discuss in details about it further in the lecture.

2. The second method to open the file is by using open( ) function.

    class_name   object_name;
    object_name.open("file_name", mode);

First we just created an object of the required class and then called the open( ) function to open the file. Parameters remain the same.

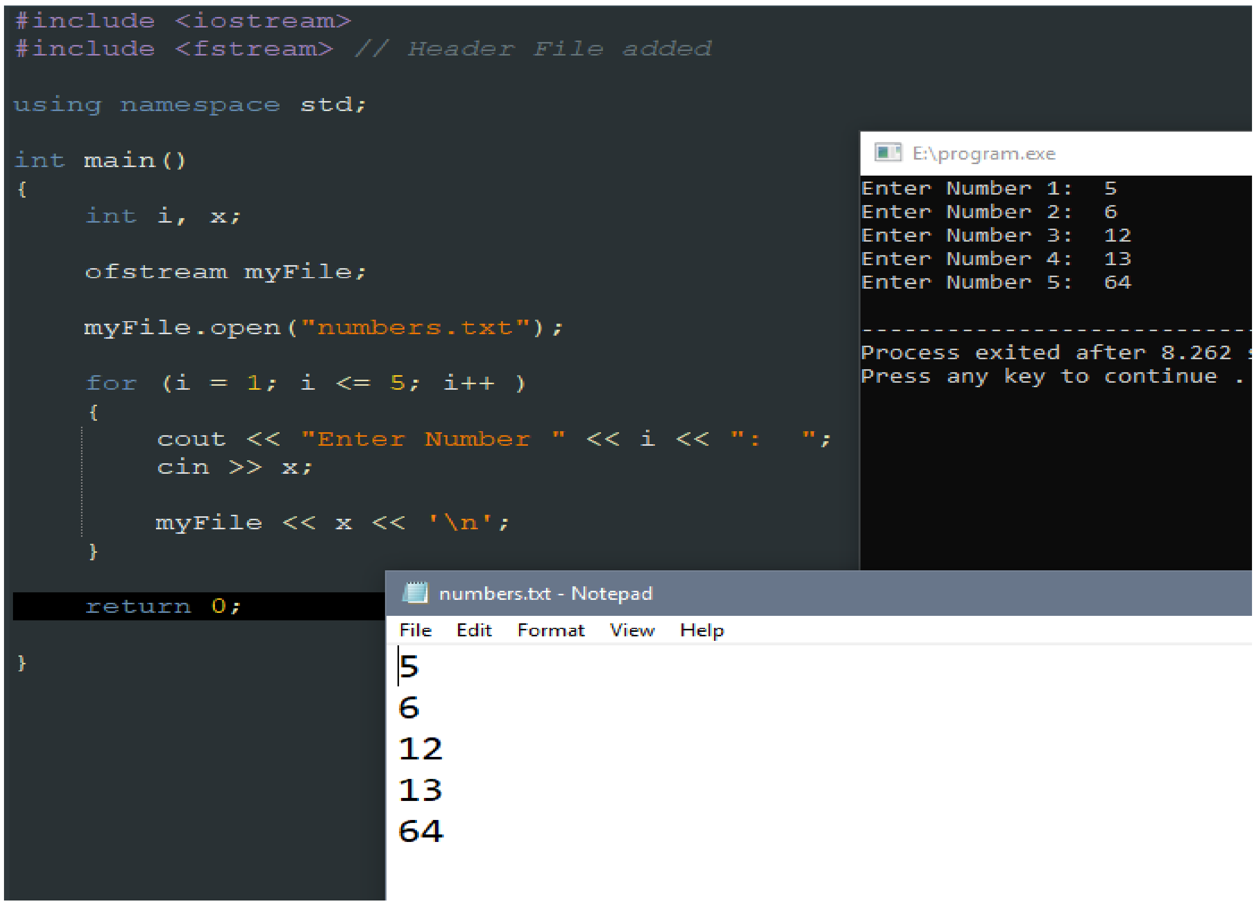
Every file which is opened needs to be closed otherwise there may be the loss of data in the file or the file can be corrupted. To close the file we can use :

object_name.close( );

It will close the file. However, we can open another file again with this object. We don't need to create the object again. We can again call the open( ) function with the same syntax to open the file. But remember that we cannot open multiple files with the same object at a time. We need to close the previous file first. Or we can make multiple objects to open multiple files in the program.

We can use the stream insertion (<<) and stream extraction(>>) operators to input and output to the file. Just like we use it with the cout and cin statements.

Sample Program to take input from user and writing in file,



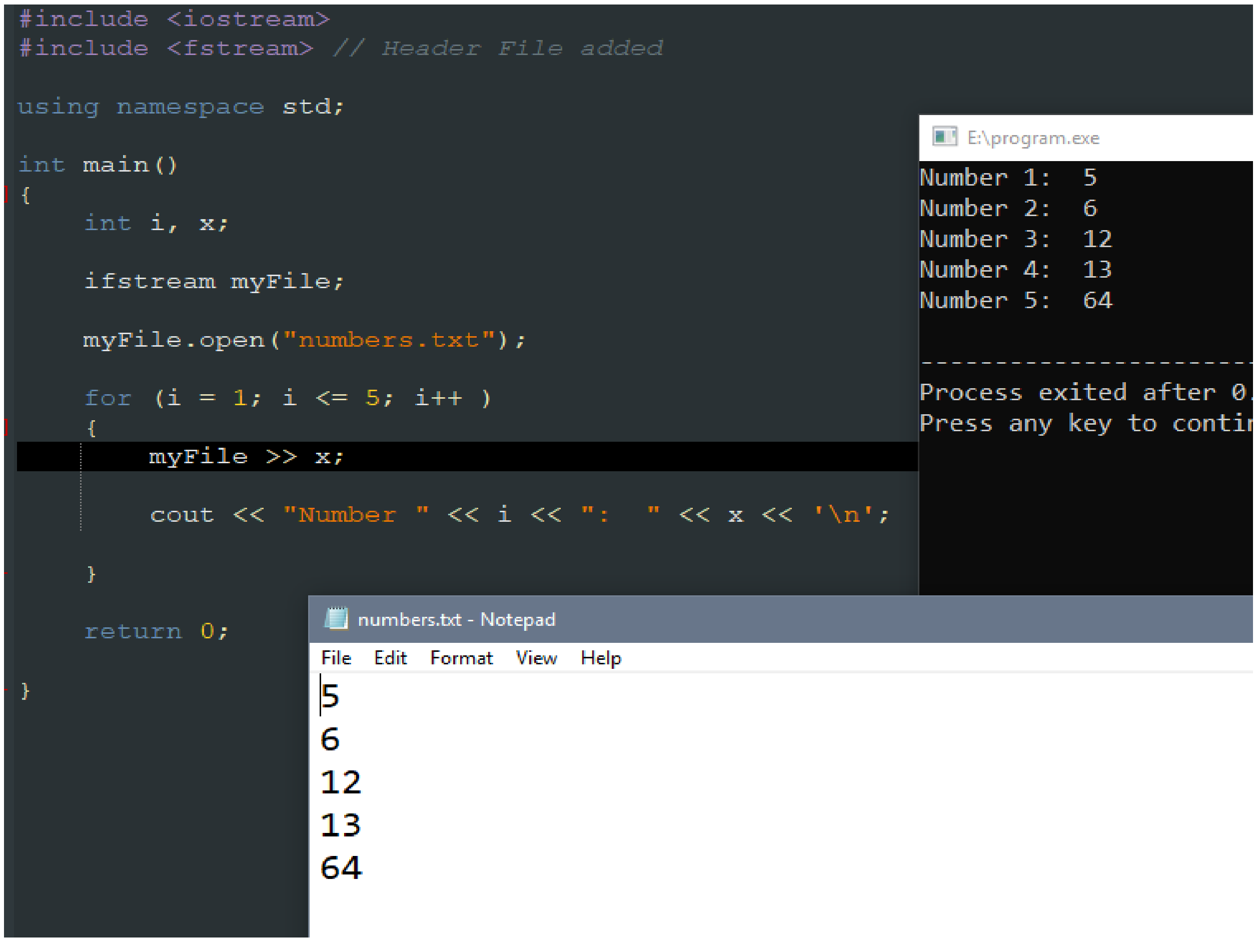
```cpp
#include <iostream>
#include <fstream> // Header File added

using namespace std;

int main()
{
    int i, x;

    ofstream myFile;

    myFile.open("numbers.txt");

    for (i = 1; i <= 5; i++ )
    {
        cout << "Enter Number " << i << ":  ";
        cin >> x;

        myFile << x << '\n';
    }

    return 0;

}
```

```
Enter Number 1:   5
Enter Number 2:   6
Enter Number 3:   12
Enter Number 4:   13
Enter Number 5:   64

--------------------------
Process exited after 8.262 s
Press any key to continue .
```

```
5
6
12
13
64
```

Edit : I forgot to close the file in both of these programs. However, we should consider it like a duty to do so in order to avoid any unexpected results. 🤔

Sample Program to take input from file printing it on the console



```cpp
#include <iostream>
#include <fstream> // Header File added

using namespace std;

int main()
{
    int i, x;

    ifstream myFile;

    myFile.open("numbers.txt");

    for (i = 1; i <= 5; i++ )
    {
        myFile >> x;

        cout << "Number " << i << ":  " << x << '\n';

    }

    return 0;

}
```

```
Number 1:  5
Number 2:  6
Number 3:  12
Number 4:  13
Number 5:  64

----------------------
Process exited after 0.
Press any key to conti
```

```
5
6
12
13
64
```

File Opening modes :

There are almost 8 file opening modes in C++, but here we will discuss only 3 but most used modes of opening files. These are :

- ios :: in —------> It used to open file for **reading** purpose only, the file must exist otherwise it cannot open the file.
- ios :: out —------> It used to open file for **writing** purpose only, A new file will be created if it does not exist before.
- ios :: app —---> It used to open file for **append** purpose only, A new file will be created if it does not exist before.

Difference Between ios :: out and ios :: app :

The ios :: out mode will truncate the file. It means that if there was any previous data in the file, it will be deleted and new data will be stored from the beginning of the file. A previous data would be lost while opening in this mode.

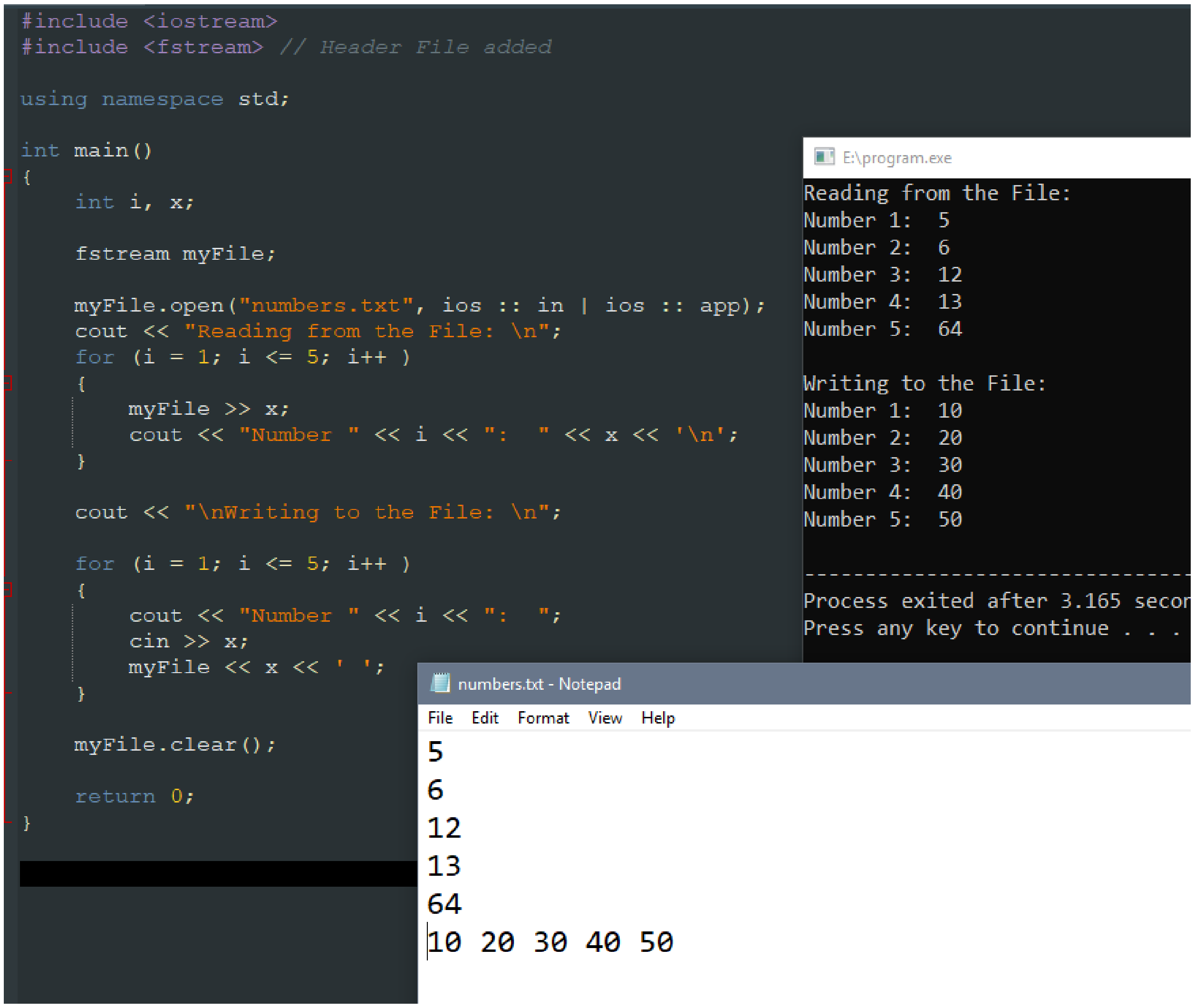
The ios :: app mode will open the file for appending data. It will not delete any previous data and starts to add the new data at the end of the previous data. It means that the previous data of the file will remain in its place and new data will be added at the end of the file.

We can open a file in multiple modes as well. For Example:

```
fstream myFile;
myFile.open("numbers.txt", ios :: in | ios :: out);
```

It will open the file “numbers.txt” in both reading and writing modes.

Sample Program :



```cpp
#include <iostream>
#include <fstream> // Header File added

using namespace std;

int main()
{
    int i, x;

    fstream myFile;

    myFile.open("numbers.txt", ios :: in | ios :: app);
    cout << "Reading from the File: \n";
    for (i = 1; i <= 5; i++ )
    {
        myFile >> x;
        cout << "Number " << i << ":  " << x << '\n';
    }

    cout << "\nWriting to the File: \n";

    for (i = 1; i <= 5; i++ )
    {
        cout << "Number " << i << ":  ";
        cin >> x;
        myFile << x << ' ';
    }

    myFile.clear();

    return 0;
}
```

```
Reading from the File:
Number 1:  5
Number 2:  6
Number 3:  12
Number 4:  13
Number 5:  64

Writing to the File:
Number 1:  10
Number 2:  20
Number 3:  30
Number 4:  40
Number 5:  50

--------------------------------
Process exited after 3.165 secor
Press any key to continue . . .
```

```
5
6
12
13
64
10 20 30 40 50
```

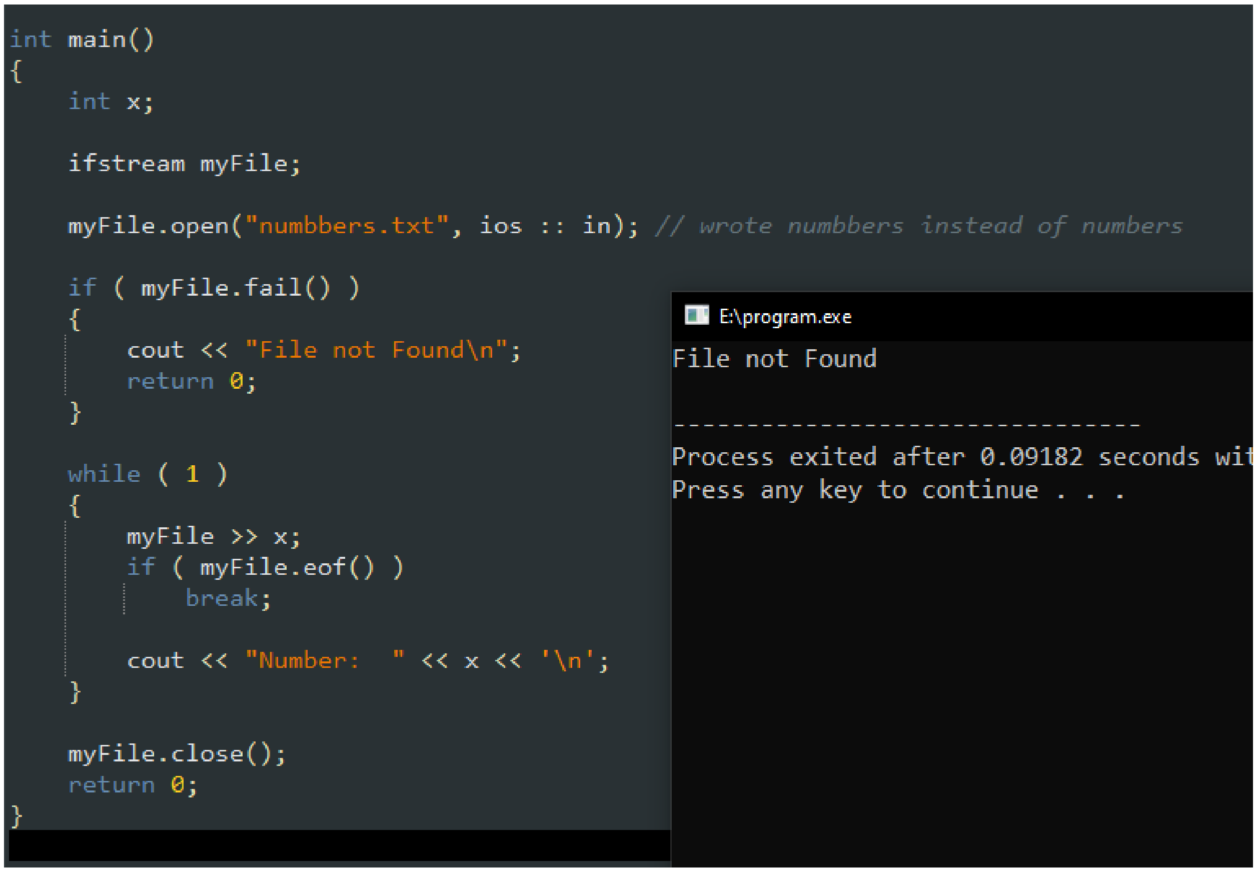
Flags associated with the File Handling in C++ :

There are certain flags that are associated with files in C++. These flags have some special functions to check their values. These are used to handle the unexpected results and used to show some special results associated with the files. Some of the important flags and their functions are as follows:

1. Fail Flag :

As we discussed earlier, the ios :: in, ios :: out and ios :: app modes are used to open files. If the file exists already it will be opened directly, but if the file does not exist the modes ios :: out and ios :: app will create a new file and open it but the ios :: in mode (which is used to read data from files) can not create new files. It will not open any file and it will set the fail flag. There is a special function fail( ) in C++ which is used to check the status of the fail flag. We can call this function by **object_name.fail( );** and it will return true if the fail flag is set, it ultimately means that the file has failed to open. Most of the time this error occurs when we pass wrong spellings or path of the file.

Sample Program : Wrote numbbers instead of numbers



```
int main()
{
    int x;

    ifstream myFile;

    myFile.open("numbbers.txt", ios :: in); // wrote numbbers instead of numbers

    if ( myFile.fail() )
    {
        cout << "File not Found\n";
        return 0;
    }

    while ( 1 )
    {
        myFile >> x;
        if ( myFile.eof() )
            break;

        cout << "Number:  " << x << '\n';
    }

    myFile.close();
    return 0;
}
```

E:\program.exe

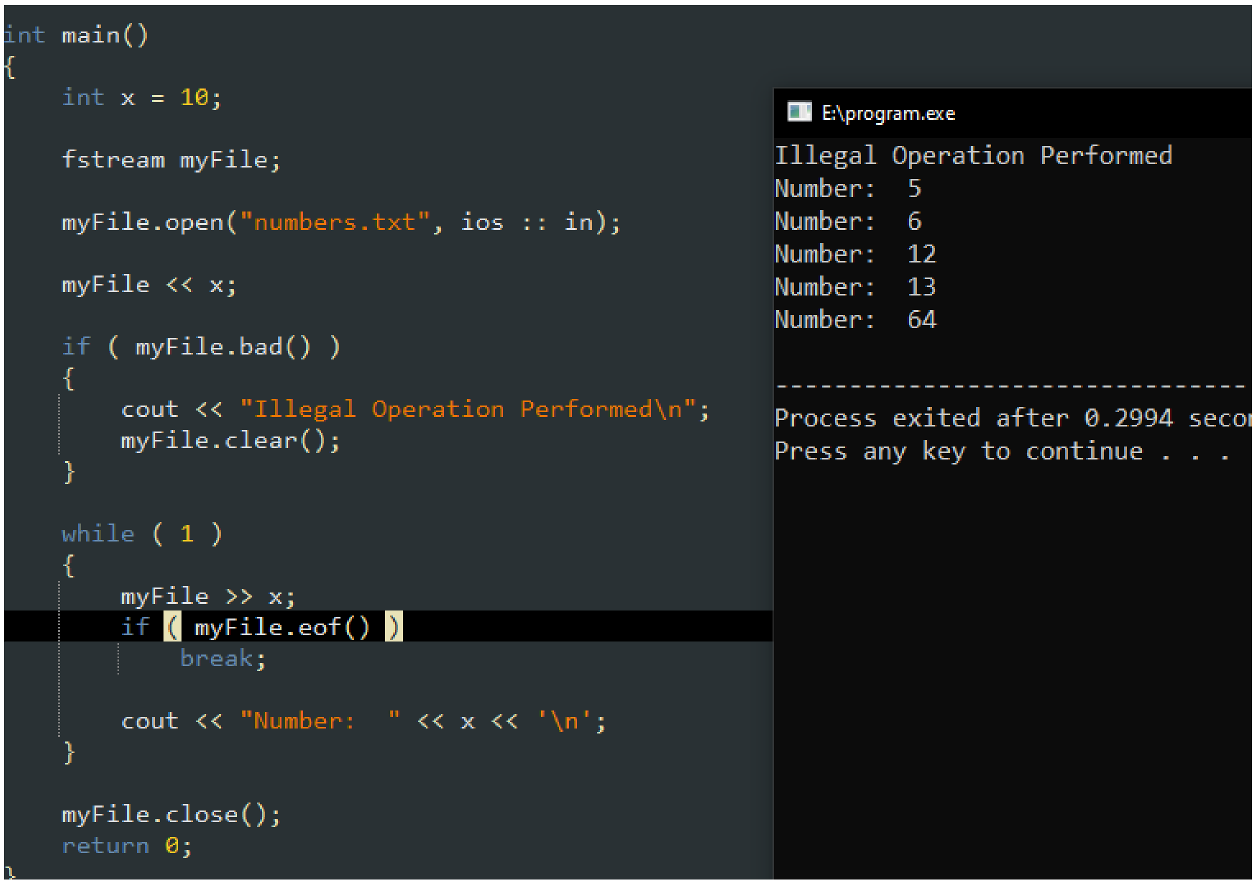
```
File not Found

--------------------------------
Process exited after 0.09182 seconds wit
Press any key to continue . . .
```

Note : The eof( ) function is discussed at the end of the flags.

2. **Bad Flag:**

The Bad flag is set when we try to do any illegal operation Like, if we try to use << (output operator) with an object which is pointing to a file which was opened in read mode only, such an illegal function will set the bad flag and we cannot process our file further, it will give unexpected results. The bad( ) function is used to check the bad flag and returns true if bad flag is set.

clear () function : The clear function is used to reset all the flags. It is mostly used with the bad flag however it can reset all the flags. When an illegal operation is performed we need to clear the flag statuses in order to work on the file correctly.



```
int main()
{
    int x = 10;

    fstream myFile;

    myFile.open("numbers.txt", ios :: in);

    myFile << x;

    if ( myFile.bad() )
    {
        cout << "Illegal Operation Performed\n";
        myFile.clear();
    }

    while ( 1 )
    {
        myFile >> x;
        if ( myFile.eof() )
            break;

        cout << "Number:  " << x << '\n';
    }

    myFile.close();
    return 0;
}
```

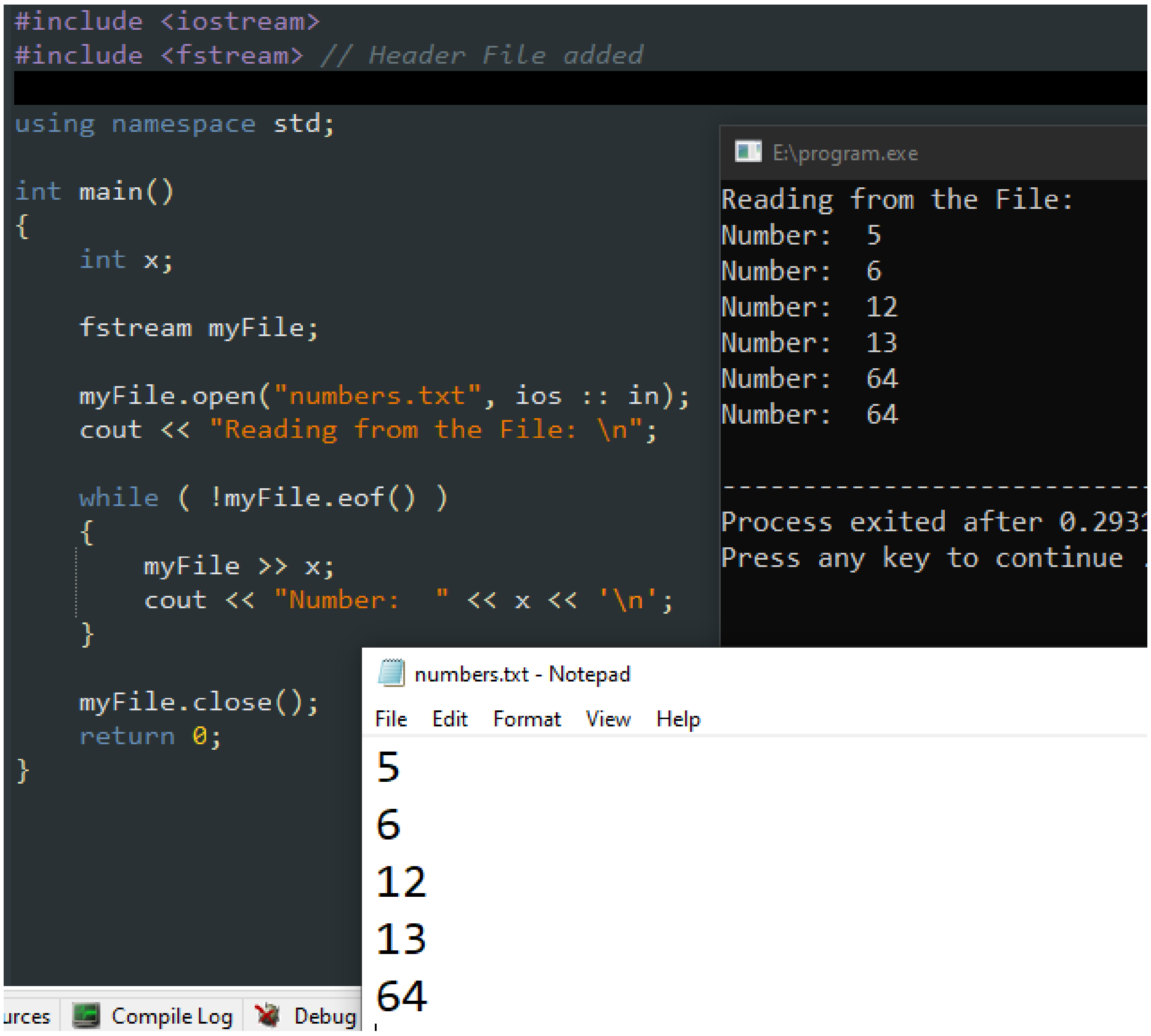
E:\program.exe

```
Illegal Operation Performed
Number:   5
Number:   6
Number:   12
Number:   13
Number:   64

--------------------------------
Process exited after 0.2994 seco
Press any key to continue . . .
```

3. **EOF Flag :**

EOF stands for End Of File. This flag is set (returns true) when the file from which we are reading data has reached its end. The amount of data stored in the file, however, is often unknown. So, do we spend our time counting data in a text file by hand, or do we let the computer deal with the amount of data? Of course, we let the computer do the counting. C++ provides a special function, **eof( )**, that returns nonzero (meaning TRUE) when there is no more data to be read from an input file stream or when the end of the file has reached.

Sample Program :



```cpp
#include <iostream>
#include <fstream> // Header File added
[redacted]
using namespace std;

int main()
{
    int x;

    fstream myFile;

    myFile.open("numbers.txt", ios :: in);
    cout << "Reading from the File: \n";

    while ( !myFile.eof() )
    {
        myFile >> x;
        cout << "Number:  " << x << '\n';
    }

    myFile.close();
    return 0;
}
```

E:\program.exe

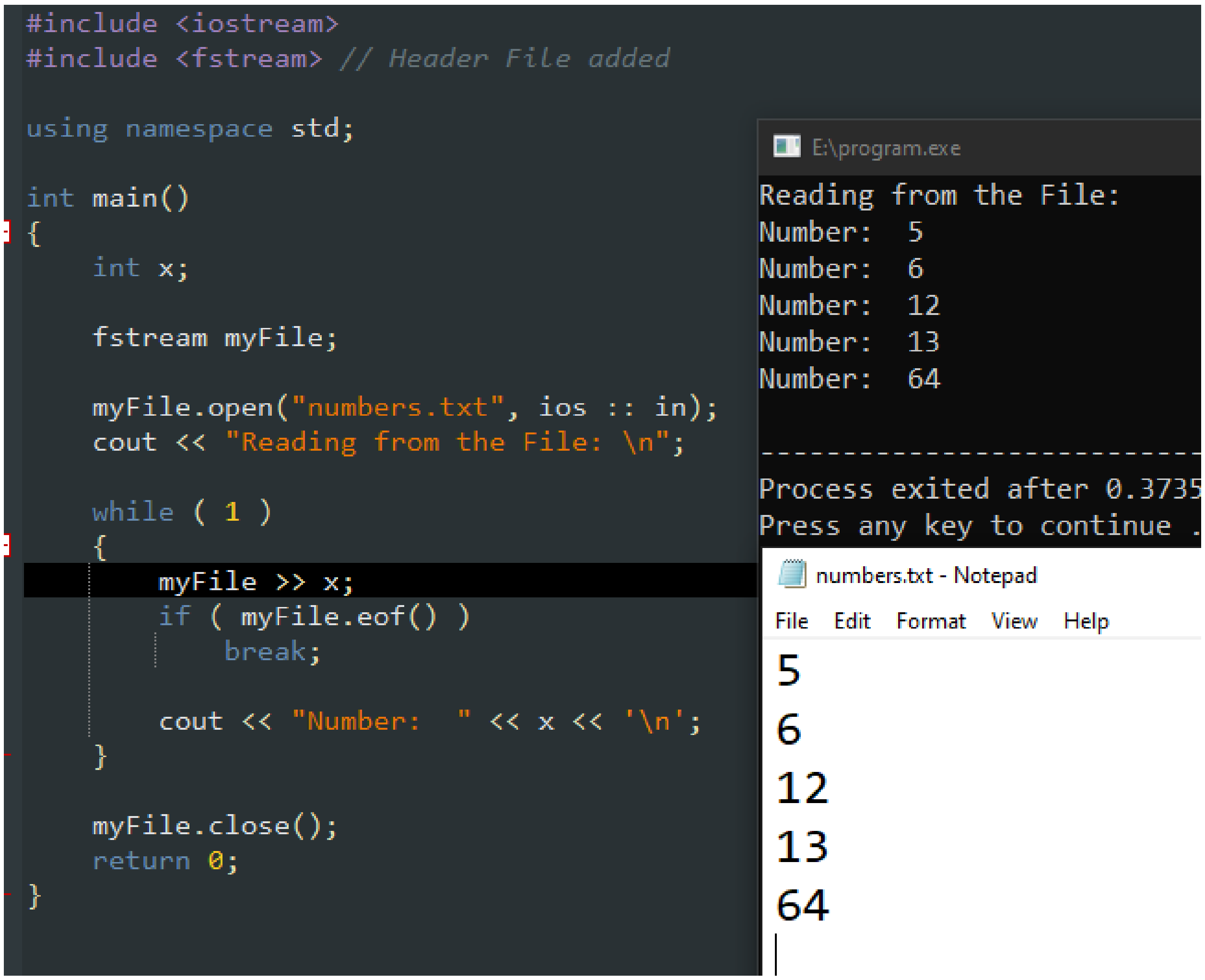
```
Reading from the File:
Number:  5
Number:  6
Number:  12
Number:  13
Number:  64
Number:  64

--------------------------
Process exited after 0.293
Press any key to continue
```

numbers.txt - Notepad

```
5
6
12
13
64
```

Why did the last Number (64) occur 2 times ?
The last number occurred two times because when the number (64) was read the first time the compiler did not know EOF had occurred. The compiler continued the loop as the EOF flag was not set. When the compiler tried to read the number after 64. It came to know that there was nothing after that and the end of file had reached. So, the compiler set the EOF flag and did not read anything into the x variable, but the very next statement is the cout statement so, it will print the previous value of x which was 64. So, how can we handle this problem? Just think a bit about it, before going further.

Solution :



```cpp
#include <iostream>
#include <fstream> // Header File added

using namespace std;

int main()
{
    int x;

    fstream myFile;

    myFile.open("numbers.txt", ios :: in);
    cout << "Reading from the File: \n";

    while ( 1 )
    {
        myFile >> x;
        if ( myFile.eof() )
            break;

        cout << "Number:  " << x << '\n';
    }

    myFile.close();
    return 0;
}
```

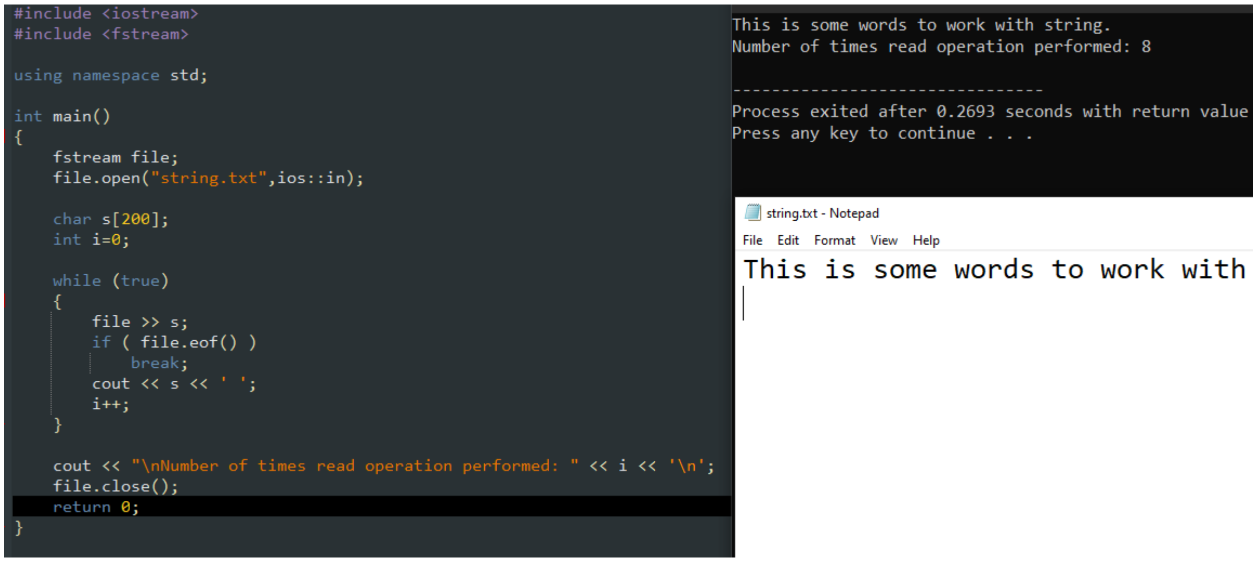
```
Reading from the File:
Number:  5
Number:  6
Number:  12
Number:  13
Number:  64

--------------------------------
Process exited after 0.3735
Press any key to continue .
```

```
5
6
12
13
64
```

Note : If a file is opened for both reading and writing. And we started reading the file and reached the end of the file. It means that the EOF flag is set. Once the EOF flag is set we cannot write new data in the file. We have to write myFile.clear( ) to reset the EOF flag. Then we can easily write new data into our file.

Strings in Files :

We can read strings as well just like we read other numbers files. Except there is one change we want to do here is that the stream extraction operator (>>) reads till the space and in strings we need to read the whole line. Check this program :



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

using namespace std;

int main()
{
    fstream file;
    file.open("string.txt",ios::in);

    char s[200];
    int i=0;

    while (true)
    {
        file >> s;
        if ( file.eof() )
            break;
        cout << s << ' ';
        i++;
    }

    cout << "\nNumber of times read operation performed: " << i << '\n';
    file.close();
    return 0;
}
```

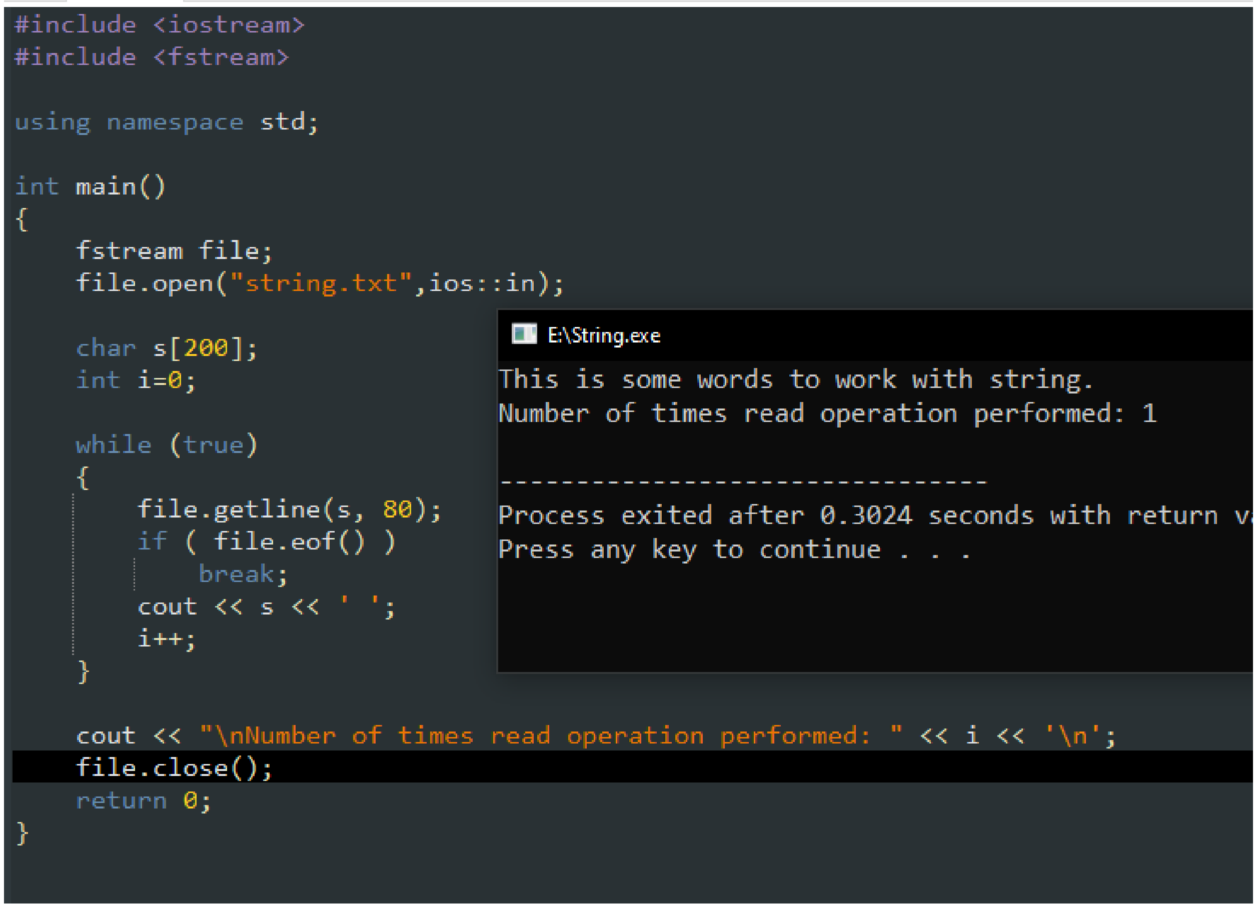
```
This is some words to work with string.
Number of times read operation performed: 8

--------------------------------
Process exited after 0.2693 seconds with return value
Press any key to continue . . .
```

```
This is some words to work with
```

We can see that the loop in this program runs 8 times to reach the end of file. And if we read the whole line at a time, we can do this in just one iteration of loop. To do this, we can use getline function like this,

myFile.getline(s, 80);

Here, s is the name of the string or character array(both will work) and the number is the maximum length of the line. This function will stop reading after it finds \n (new line) or the maximum length of the character is reached.



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

using namespace std;

int main()
{
    fstream file;
    file.open("string.txt",ios::in);

    char s[200];
    int i=0;

    while (true)
    {
        file.getline(s, 80);
        if ( file.eof() )
            break;
        cout << s << ' ';
        i++;
    }

    cout << "\nNumber of times read operation performed: " << i << '\n';
    file.close();
    return 0;
}
```

```
This is some words to work with string.
Number of times read operation performed: 1

--------------------------------
Process exited after 0.3024 seconds with return v
Press any key to continue . . .
```

Moving through data in a file :

There are several functions that are used to move through the data. These are seekg(), seekp(), tellg() and tellp(). seekg() and seekp() both are functions of File Handling in C++ and they are very important and useful features of File Handling in C++. In File Handling of C++, we have two pointers: one is get pointer and second is put pointer. They are used to control the positions of where we want to read the data from a file and where to write the data into a file.

seekg() and tellg() :

seekg() function is used to move the get pointer at the desired position to start reading data at that point from a file . If seekg() function is used to move the get pointer at the particular location. So then by using the tellg() function we easily get the current position of the get pointer from the external file.

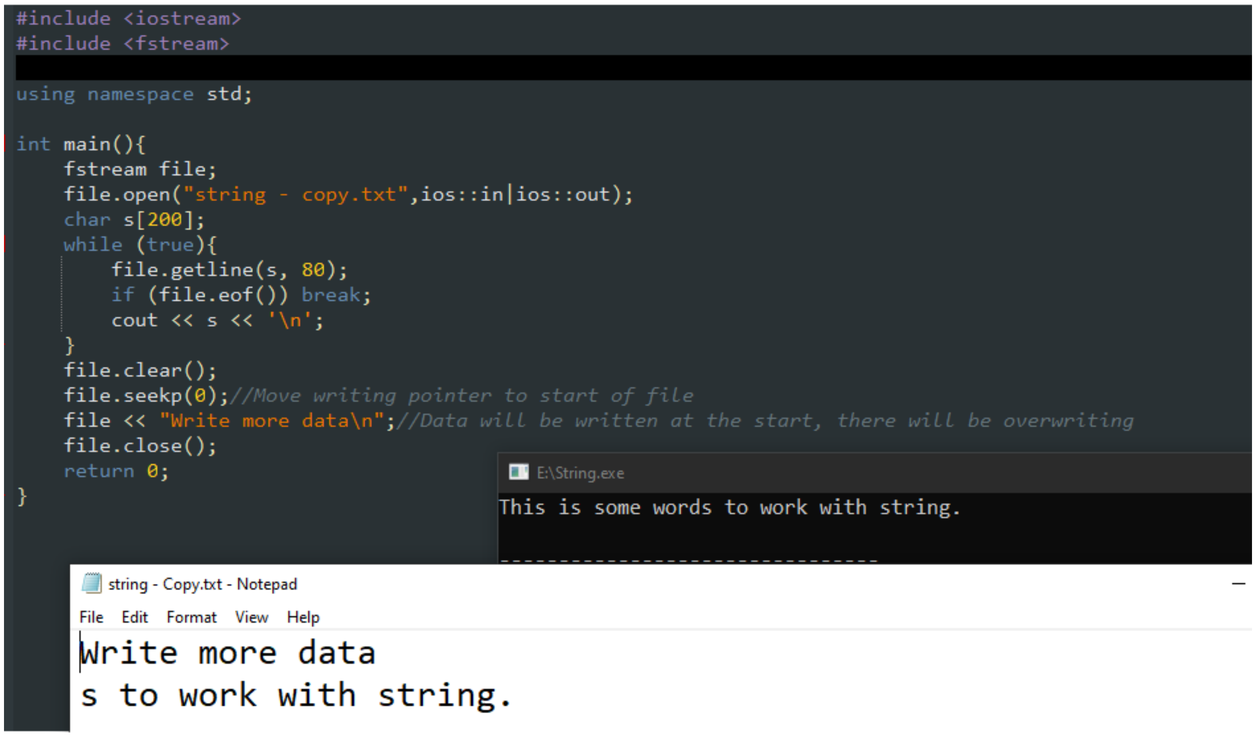
Syntax: seekg(no of bytes to move in the file);

seekp() and tellp() :

seekp() function is used to move the put pointer at the particular location to start writing/putting the data at that point into a file. And If seekp() function is used to move the put pointer at the particular/desired location. So then by using the tellp() function we easily get the current position of the put pointer from the file.

Syntax: seekp (no of bytes to move in the file);

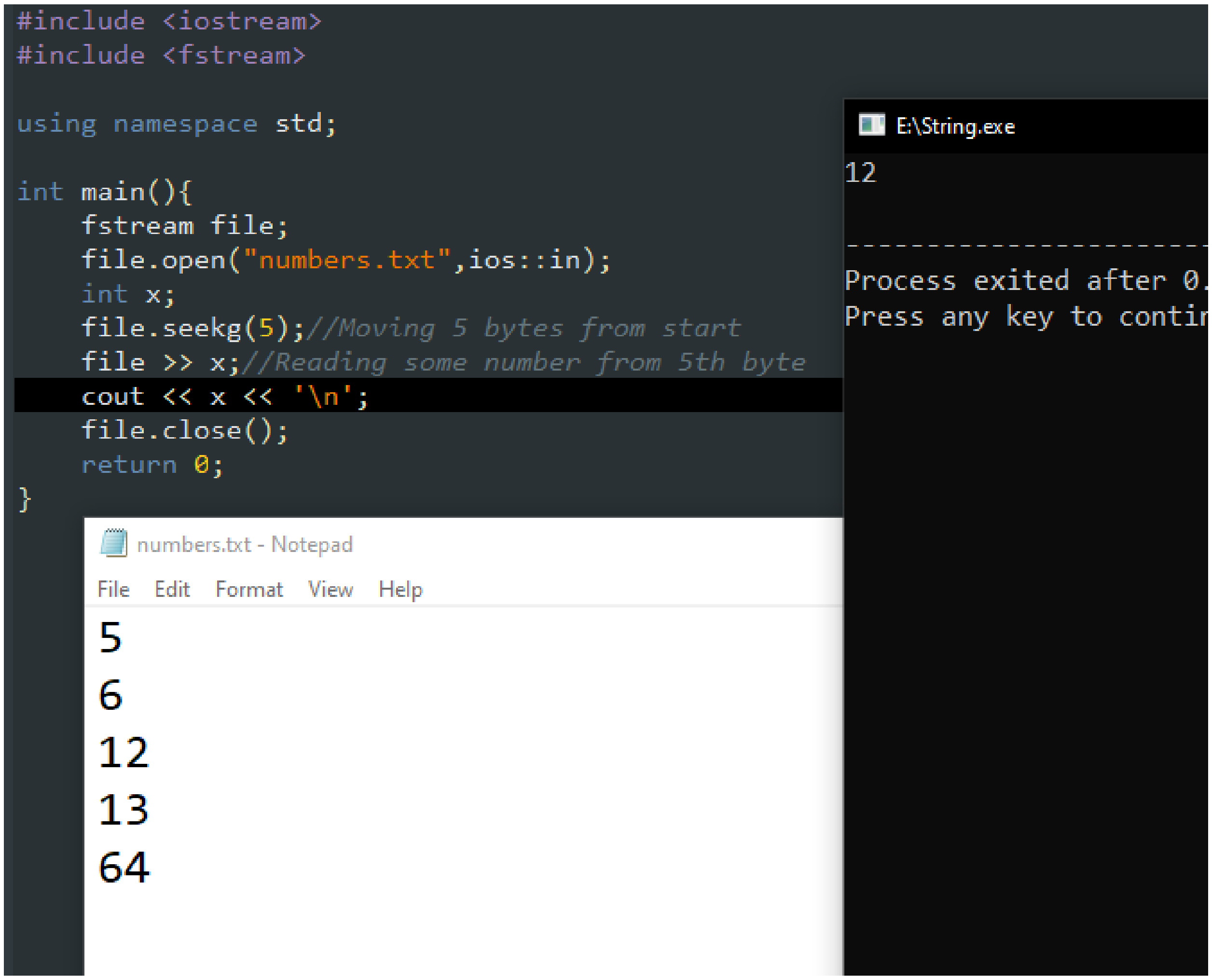
Sample Programs :



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

using namespace std;

int main(){
    fstream file;
    file.open("string - copy.txt",ios::in|ios::out);
    char s[200];
    while (true){
        file.getline(s, 80);
        if (file.eof()) break;
        cout << s << '\n';
    }
    file.clear();
    file.seekp(0);//Move writing pointer to start of file
    file << "Write more data\n";//Data will be written at the start, there will be overwriting
    file.close();
    return 0;
}
```

```
This is some words to work with string.

--------------------------------
```

```
Write more data
s to work with string.
```



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

using namespace std;

int main(){
    fstream file;
    file.open("numbers.txt",ios::in);
    int x;
    file.seekg(5);//Moving 5 bytes from start
    file >> x;//Reading some number from 5th byte
    cout << x << '\n';
    file.close();
    return 0;
}
```

```
12

------------------------
Process exited after 0.
Press any key to contin
```

```
5
6
12
13
64
```

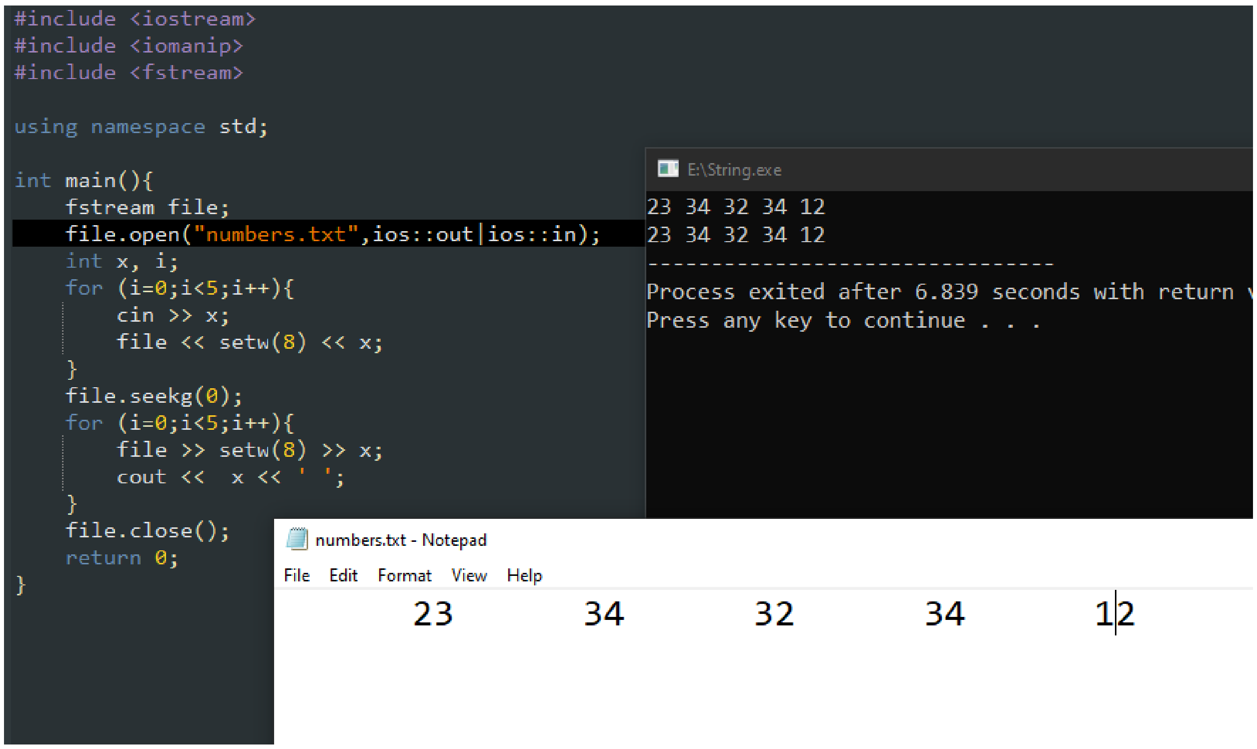
Here, we moved 5 bytes from the start. In text files, every number is stored as characters. It means that it will take 1 byte in memory. So, in this program the first byte is 5, second byte is line break i.e \n (\n is also a character with ascii value 13), third byte is 6 and forth byte is again \n. The fifth byte is 1 but as we are writing in an int variable, it will read the whole number before the next space or line break, hence it reads 12 and displays it on the screen.

Note : We need to add header file <iomanip> to use setw() and setprecision() functions.

setw() :

setw() stands for set width. It is used to set the maximum width for any number. It means that it will specify the fixed amount of length for each number. As 2 is 1 digit number and 1000 is 4 digit number and they will be stored at different locations and both will take different numbers of bytes in the file. So, if we want to move through the data and every number is taking different amounts of bytes, it will be difficult to count how many bytes we have to jump. If we use setw(4); function before writing data in the file it will store every number in fixed width, it means that 2 will be stored with 3 blank spaces at the start 20 will be stored with 2 blank spaces and 1000 will be stored directly in the file. It will help us to move through the data when every number is taking a fixed amount of bytes. So, in this case if we want to jump 3 numbers and read 4th number we can simple write seekg(4*3); here, 4 is the width of every number and 3 tells us that we have to jump 3 numbers and start reading after the 3rd number.

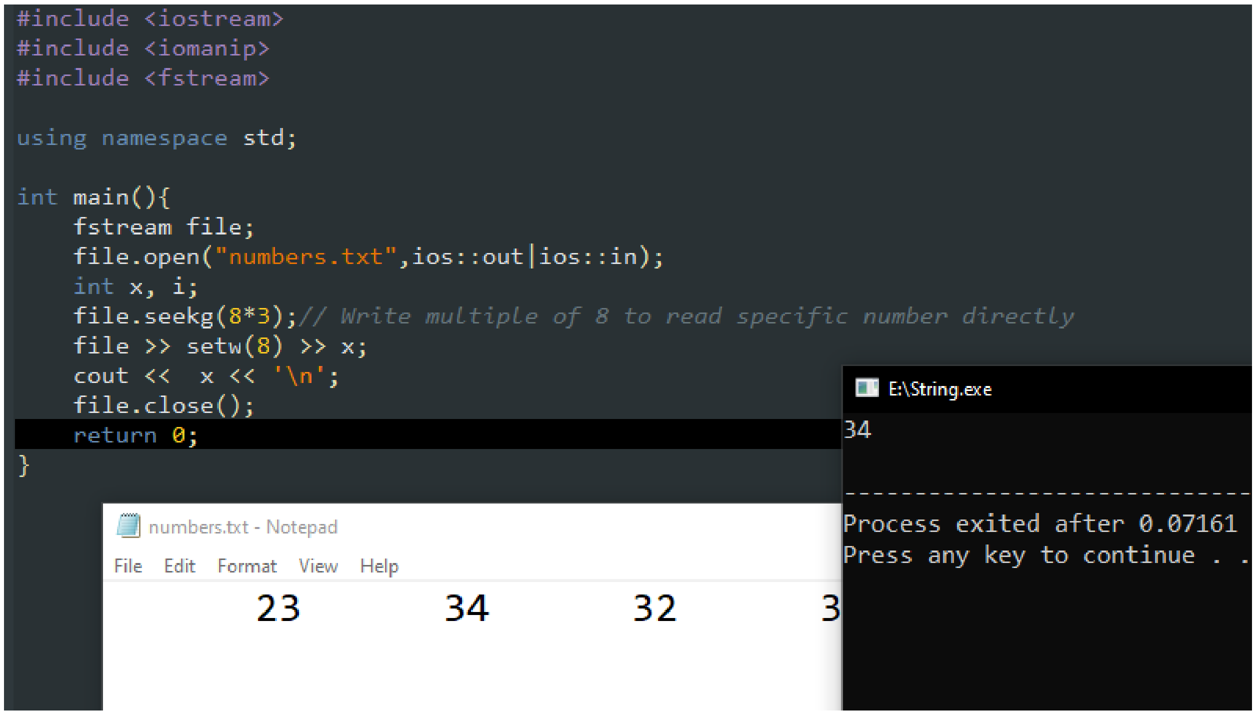
Sample Programs :



```cpp
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

int main(){
    fstream file;
    file.open("numbers.txt",ios::out|ios::in);
    int x, i;
    for (i=0;i<5;i++){
        cin >> x;
        file << setw(8) << x;
    }
    file.seekg(0);
    for (i=0;i<5;i++){
        file >> setw(8) >> x;
        cout <<  x << ' ';
    }
    file.close();
    return 0;
}
```

```
23 34 32 34 12
23 34 32 34 12
--------------------------------
Process exited after 6.839 seconds with return v
Press any key to continue . . .
```

```
      23      34      32      34      12
```



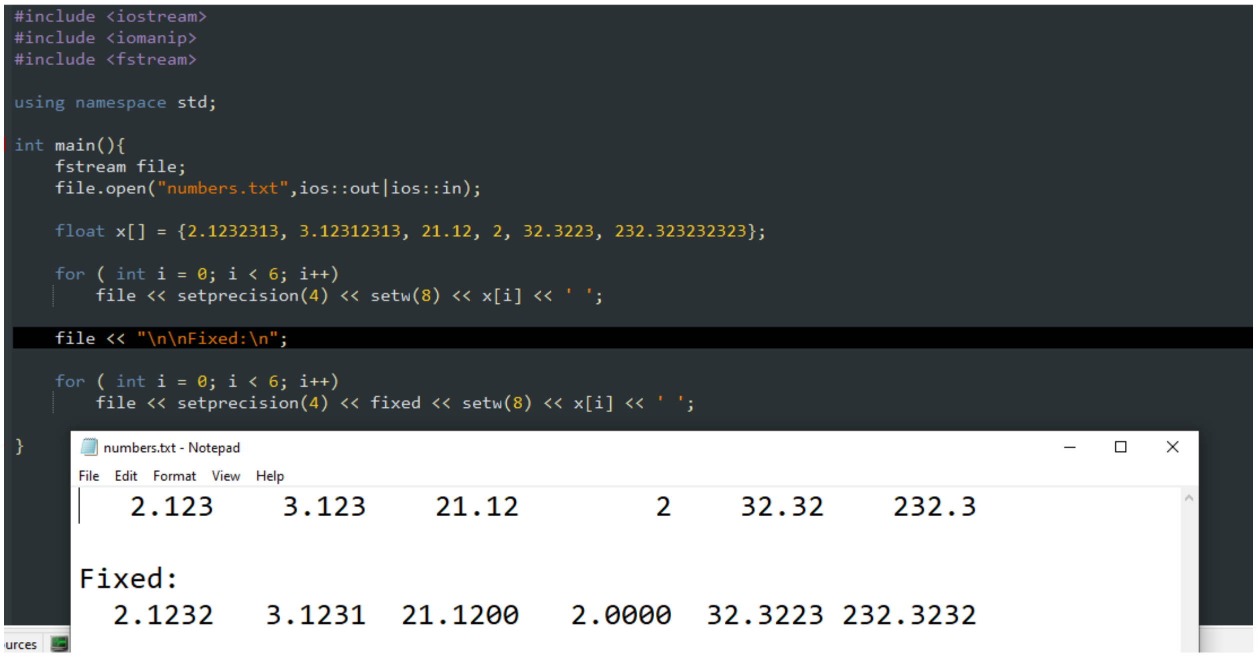
```
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

int main(){
    fstream file;
    file.open("numbers.txt",ios::out|ios::in);
    int x, i;
    file.seekg(8*3);// Write multiple of 8 to read specific number directly
    file >> setw(8) >> x;
    cout <<  x << '\n';
    file.close();
    return 0;
}
```

```
        23        34        32        3
```

```
34

------------------------------
Process exited after 0.07161
Press any key to continue . .
```

setprecision() :

setprecision(); function is used to control the output of floating point numbers. Setprecision sets the maximum numbers that will occur after the decimal point. However, if we use **fixed** keyword along with the setprecision() it will set both the minimum and maximum number of digits after decimal and fill empty spaces with 0.



```cpp
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

int main(){
    fstream file;
    file.open("numbers.txt",ios::out|ios::in);

    float x[] = {2.1232313, 3.12312313, 21.12, 2, 32.3223, 232.323232323};

    for ( int i = 0; i < 6; i++)
        file << setprecision(4) << setw(8) << x[i] << ' ';

    file << "\n\nFixed:\n";

    for ( int i = 0; i < 6; i++)
        file << setprecision(4) << fixed << setw(8) << x[i] << ' ';

}
```

```
   2.123    3.123    21.12        2    32.32    232.3

Fixed:
  2.1232   3.1231  21.1200   2.0000  32.3223 232.3232
```

-THE END-