All About: File I/O in C++

By **Ilia Yordanov**, loobian@cpp-home.com
www.cpp-home.com ; C++ Resources

Introduction

This tutorial will start with the very basis of File I/O (Input/Output) in C++. After that, I will look into aspects that are more advanced, showing you some tricks, and describing useful functions.
You need to have good understanding of C++, otherwise this tutorial will be unfamiliar and not useful to you!

Your Very First Program

I will first write the code, and after that, I will explain it line by line.
The first program, will create a file, and put some text into it.

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

int main()
{
      ofstream SaveFile("cpp-home.txt");
      SaveFile << "Hello World, from www.cpp-home.com and Loobian!";
      SaveFile.close();
      return 0;
}
```

Only that? Yes! This program will create the file cpp-home.txt in the directory from where you are executing it, and will put "Hello World, from www.cpp-home.com and Loobian!" into it.
Here is what every line means:

`#include <fstream>` - You need to include this file in order to use C++'s functions for File I/O.
In this file, are declared several classes, including ifstream, ofstream and fstream, which are all derived from istream and ostream.

`ofstream SaveFile("cpp-home.txt");`

1) ofstream means "output file stream". It creates a handle for a stream to write in a file.
2) SaveFile – that's the name of the handle. You can pick whatever you want!
3) ("cpp-home.txt"); - opens the file cpp-home.txt, which should be placed in the directory from where you execute the program. If such a file does not exists, it will be created for you, so you don't need to worry about that!

Now, let's look a bit deeper. First, I'd like to mention that ofstream is a class. So, ofstream SaveFile("cpp-home.txt"); creates an object from this class. What we pass in the brackets, as parameter, is actually what we pass to the constructor. It is the name of the file. So, to summarize: we create an object from class ofstream, and we pass the name of the file we want to create, as an argument to the class' constructor. There are other things, too, that we can pass, but I will look into that, later.

`SaveFile << "Hello World, from www.cpp-home.com and Loobian!";` - "<<" looks familiar? Yes, you've seen it in cout <<. This ("<<") is a predefined operator. Anyway, what this line makes, is to put the text above in the file. As mentioned before, SaveFile is a handle to the opened file stream. So, we write the handle name, << and after it we write the text in inverted commas. If we want to pass variables instead of text in inverted commas, just pass it as a regular use of the cout <<. This way:
`SaveFile << variablename;`
That's it!

`SaveFile.close();` - As we have opened the stream, when we finish using it, we have to close it. SaveFile is an object from class ofstream, and this class (ofstream) has a function that closes the stream. That is the close() function. So, we just write the name of the handle, dot and close(), in order

to close the file stream!
Notice: Once you have closed the file, you can't access it anymore, until you open it again.

That's the simplest program, to write in a file. It's really easy! But as you will see later in this tutorial, there are more things to learn!

Reading A File

You saw how to write into a file. Now, when we have cpp-home.txt, we will read it, and display it on the screen.
First, I'd like to mention, that there are several ways to read a file. I will tell you about all of them (all I know) later. For now, I will show you the best way (in my mind).
As you are used already- I will first write the code, and after that, I will comment it in details.

```
#include <fstream.h>

void main() //the program starts here
{
	ifstream OpenFile("cpp-home.txt");

	char ch;
	while(!OpenFile.eof())
	{
		OpenFile.get(ch);
		cout << ch;
	}
	OpenFile.close();
}
```

You should already know what the first line is. So, let me explain you the rest.

`ifstream OpenFile("cpp-home.txt")` – I suppose this seems a bit more familiar to you, already! ifstream means "input file stream". In the previous program, it was ofstream, which means "output file stream". The previous program is to write a file, that's why it was "output". But this program is to read from a file, that's why it is "input". The rest of the code on this line, should be familiar to you. OpenFile is the object from class ifstream, which will handle the input file stream. And in the inverted commas, is the name of the file to open.
Notice that that there is no check whether the file exists! I will show you how to check that, later!

`char ch;` - Declares an array of type char. Just to remind you- such arrays can hold just one sign from the ASCII table.

`while(!OpenFile.eof())` – The function eof() returns a nonzero value if the end of the file has been reached. So, we make a while loop, that will loop until we reach the end of the file. So, we will get through the whole file, so that we can read it!

`OpenFile.get(ch);` - OpenFile is the object from class ifstream. This class declares a function called get(). So, we can use this function, as long as we have an object. The get() function extracts a single character from the stream and returns it. In this example, the get() function takes just one parameter- the variable name, where to put the read character. So, after calling OpenFile.get(ch) it will read one character from the stream OpenFile, and will put this character into the variable ch.
Notice: If you call this function for a second time, it will read the next character, but not the same one! You will learn why this happens, later.
That's why, we loop until we reach the end of the file! And every time we loop, we read one character and put it into ch.

`cout <<  ch;` - Display ch, which has the read character.

`File.close();` - As we have opened the file stream, we need to close it. Use the close() function, to close it! Just as in the previous program!
Notice: Once you have closed the file, you can't access it anymore, until you open it again.

That's all! I hope you understood my comments! When you compile and run this program, it should output:

Managing I/O streams

In this chapter, I will mention about some useful functions. I will also show you how to open file to read and write in the same time. I will show you, also, other ways to open a file; how to check if opening was successful or not. So- read on!

So far, I have showed to you, just one way to open a file, either for reading, either for writing. But it can be opened another way, too! So far, you should be aware of this method:

```
ifstream OpenFile(“cpp-home.txt”);
```

Well, this is not the only way! As mentioned before, the above code creates an object from class ifstream, and passes the name of the file to be opened to its constructor. But in fact, there are several overloaded constructors, which can take more than one parameter. Also, there is function open() that can do the same job. Here is an example of the above code, but using the open() function:

```
ifstream OpenFile;
OpenFile.open(“cpp-home.txt”);
```

What is the difference you ask? Well, I made several tests, and found no difference! Just if you want to create a file handle, but don’t want to specify the file name immediately, you can specify it later with the function open(). And by the way, other use of open() is for example if you open a file, then close it, and using the same file handle open another file. This way, you will need the open() function. Consider the following code example:

```
#include <fstream.h>

void read(ifstream &T) //pass the file stream to the function
{
      //the method to read a file, that I showed you before
      char ch;

      while(!T.eof())
      {
            T.get(ch);
            cout << ch;
      }

      cout << endl << "--------" << endl;
}

void main()
{
      ifstream T("file1.txt");
      read(T);
      T.close();

      T.open("file2.txt");
      read(T);
      T.close();
}
```

So, as long as file1.txt and file2.txt exists and has some text into, you will see it!

Now, it’s time to show you that the file name is not the only parameter that you can pass to the open() function or the constructor (it’s the same). Here is a prototype:

```
ifstream OpenFile(char *filename, int open_mode);
```

You should know that filename is the name of the file (a string). The new here is the open mode. The value of open_mode defines how to be opened the file. Here is a table of the open modes:

Name	Description
ios::in	Open file to read
ios::out	Open file to write
ios::app	All the date you write, is put at the end of the file. It calls ios::out
ios::ate	All the date you write, is put at the end of the file. It does not call ios::out
ios::trunc	Deletes all previous content in the file. (empties the file)
ios::nocreate	If the file does not exists, opening it with the open() function gets impossible.
ios::noreplace	If the file exists, trying to open it with the open() function, returns an error.
ios::binary	Opens the file in binary mode.

In fact, all these values are int constants from an enumerated type. But for making your life easier, you can use them as you see them in the table.
Here is an example on how to use the open modes:

```
#include <fstream.h>

void main()
{
	ofstream SaveFile("file1.txt", ios::ate);

	SaveFile << "That's new!\n";

	SaveFile.close();
}
```

As you see in the table, using ios::ate will write at the end of the file. If I didn't use it, the file will be overwritten, but as I use it, I just add text to it. So, if file1.txt has this text:

Hi! This is test from www.cpp-home.com!

Running the above code, will add "That's new!" to it, so it will look this way:

Hi! This is test from www.cpp-home.com!That's new!

If you want to set more than one open mode, just use the **OR** operator- **|**. This way:

```
ios::ate | ios::binary
```

I hope you now understand what open modes are!

Now, it's time to show you something really useful! I bet you didn't know that you could create a file stream handle, which you can use to read/write file, in the same time! Here is how it works:

```
fstream File("cpp-home.txt",ios::in | ios::out);
```

In fact, that is only the declaration. I will show you a code example, just several lines bellow. But I first want to mention some things you should know.
The code line above, creates a file stream handle, named "File". As you know, this is an object from class fstream. When using fstream, you should specify ios::in and ios::out as open modes. This way, you can read from the file, and write in it, in the same time, without creating new file handles. Well, of course, you can only read or write. Then you should use either ios::in either ios::out, but if you are going to do it this way, why don't you do it either with ifstream, either with ofstream?
Here is the code example:

```
#include <fstream.h>

void main()
{
      fstream File("test.txt",ios::in | ios::out);

      File << "Hi!"; //put “Hi!” in the file

      static char str[10]; //when using static, the array is automatically
                           //initialized, and very cell NULLed

      File.seekg(ios::beg); //get back to the beginning of the file
                            //this function is explained a bit later
      File >> str;
      cout << str << endl;

      File.close();
}
```

Okay, there are some new things here, so I will explain line by line:

`fstream File(“test.txt”, ios::in | ios::out);` - This line, creates an object from class fstream. At the time of execution, the program opens the file test.txt in read/write mode. This means, that you can read from the file, and put data into it, at the same time.

`File << “Hi!”;` - I beg you know what this is!

`static char str[10];` - This makes a char array with 10 cells. I suppose static may be unfamiliar to you. If so- ignore it. It just initializes the array when at the time of creation.

`File.seekg(ios::beg);` - Okay, I want you to understand what this really do, so I will start with something a bit off-topic, but important.
Remember that? :

```
      while(!OpenFile.eof())
      {
            OpenFile.get(ch);
            cout << ch;
      }
```

Did you ever wonder what really happens there? Yes or no, I will explain you. This is a while loop, that will loop until you reach the end of the file. But how do the loop know if the end of the file is reached? Well, when you read the file, there is something like an inside-pointer, that shows where you are up to, with the reading (and writing, too). It is like the cursor in Notepad. And every time you call OpenFile.get(ch) it returns the current character to the ch variable, and moves the inside-pointer one character after that, so that the next time this function is called, it will return the next character. And this repeats, until you reach the end of the file.
So, let’s get back to the code line. The function seekg() will put the inside-pointer to a specific place (specified by you). You can use:
`ios::beg` – to put it in the beginning of the file
`ios::end` - to put it at the end of the file
Or you can also set the number of characters to go back or after. For example, if you want to go 5 characters back, you should write:

```
File.seekg(-5);
```

If you want to go 40 character after, just write:

```
File.seekg(40);
```

I also have to mention, that the seekg() function is overloaded, and it can take two parameters, too. The other version is this one:

```
File.seekg(-5,ios::end);
```

In this example, you will be able to read the last 4 characters of the text, because:

1) You go to the end (ios::end)
2) You go 5 characters before the end (-5)

Why you will read 4 but not 5 characters? Well, just assume that one is lost, because the last thing in the file is not a character nor white space. It is just position.

You now may be wondering why did I use this function? Well, after I put “Hi!” in the file, the inside-pointer was set after it… at the end of the file. And as I want to read the file, I have nothing to read after the end, so I have to put the inside-pointer at the beginning. And that is exactly what this function does.

`File >> str;` - That’s new, too! Well, I believe this line reminds you of cin >> . I fact, it has much to do with it. This line reads one word from the file, and puts it into the specified array.
For example, if the file has this text:

Hi! Do you know me?

Using File >> str, will put just “Hi!” to the str array. You should have noticed, that it actually reads until it meets a white space.
And as what I put in the file was “Hi!” I don’t need to do a while loop, that takes more time to code. That’s why I used this way. By the way, in the while loop for reading, that I used so far, the program reads the file, char by char. But you can read it word by word, this way:

```
      char str[30]; //the word can’t be more than 30 characters long
      while(!OpenFile.eof())
      {
            OpenFile >> str;
            cout << str;
      }
```

You can also read it line by line, this way:

```
      char line[100]; //a whole line will be stored here
      while(!OpenFile.eof())
      {
            OpenFile.getline(line,100); //where 100 is the size of the array
            cout << line << endl;
      }
```

You now might be wondering which way to use? Well, I’d recommend you to use the line-by-line one, or the first that I mentioned- the one which reads char-by-char. The one that reads word-by-word is not good idea, because it won’t read the new line. So if you have new line in the file, it will not display it as a new line, but will append the text to the existing one. But using getline() or get() will show you the file, just as it is!

Now, I will show you how to check if the file opening was successful or not. In fact, there are few good ways to check for that, and I will mention them. Notice that where there is X, it can be either “o”, either “i” either nothing (it will then be fstream object).

Example 1: The most usual way

```
Xfstream File(“cpp-home.txt”);
if (!File)
{
      cout << “Error opening the file! Aborting…\n”;
      exit(1);
}
```

Example 2: If the file is created, return an error

```
ofstream File("unexisting.txt", ios::nocreate);

if(!File)
{
      cout << “Error opening the file! Aborting…\n”;
      exit(1);
}
```

Example 3: Using the fail() function

```
ofstream File("filer.txt", ios::nocreate);

if(File.fail())
{
      cout << "Error opening the file! Aborting…\n";
      exit(1);
}
```

The new in Example 3, is the fail() function. It returns a nonzero value if any I/O error (not end of file) has occurred.

I would also like to mention about something , that I find to be very useful! For example, if you have created a file stream, but you haven't opened a file. This way:

```
ifstream File; //it could also be ofstream
```

This way, we have a handle, but we still have not opened the file. If you want to open it later, it can be done with the open() function, which I already covered in this tutorial. But if anywhere in your program, you need to know if currently there is an opened file, you can check it with the function is_open(). It retunrs 0 (false) if a file is not opened, and 1 (true) if there is an opened file. For example:

```
ofstream File1;
File1.open("file1.txt");
cout << File1.is_open() << endl;
```

The code above, will return 1, as we open a file (on line 2). But the code bellow will return 0, because we don't open a file, but just create a file stream handle:

```
ofstream File1;
cout << File1.is_open() << endl;
```

Okay, enough on this topic.

Checking the I/O status- Flags

I won't be explaining what flags are. But even if you don't know about them, reading this chapter to the end may give you some idea, and I believe you will understand the theory. Even so, if you don't know about the flags in C++, I recommend you to find some reading on this subject.
Okay, let's start!

The Input/Output system in C++, holds information about the result of every I/O operation. The current status is kept in an object from type **io_state**, which is an enumerated type (just like open_mode) that has the following values:

godbit	No errors.
eofbit	End of file has been reached
failbit	Non-fatal I/O error
badbit	Fatal I/O error

There are two ways, to receive information about the I/O status. One of them is by calling the function rdstate(). It returns the current status of the error-flags (the above mentioned). For example, the rdstate() function will return goodbit if there were no errors.
The other way to check the I/O status is by using any of the following function:

```
bool bad();
bool eof(); //remember this one? "Read until the end of the file has been reached!"
bool fail(); //and this one, too… Check if the file opening was successfull
bool good();
```

The function bad() returns true, if the badbit flag is up. The fail() function returns true if the failbit flag is up. The good() function returns true if there were no errors (the goodbit flag is up). And the eof() function returns true if the end of the file has been reached (the eofbit flag is up).

If an error occurred, you will have to clear it if you want your program to continue properly. To do so, use the clear() function, which takes one parameter. The parameter should be the flag you want to put to be the current status. If you want your program to continue "on clear", just put ios::goodbit as parameter. But notice that the clear() function can take any flag as parameter. You will see that in the code examples bellow.

I will now show you some example code that will confirm your knowledge.

Example 1: Simple status check

```
	//Replace FileStream with the name of the file stream handle
	if(FileStream.rdstate() == ios::eofbit)
		cout << "End of file!\n";
	if(FileStream.rdstate() == ios::badbit)
		cout << "Fatal I/O error!\n";
	if(FileStream.rdstate() == ios::failbit)
		cout << "Non-fatal I/O error!\n";
	if(FileStream.rdstate() == ios::goodbit)
		cout << "No errors!\n";
```

Example 2: The clear() function

```
#include <fstream.h>

void main()
{
	ofstream File1("file2.txt"); //create file2.txt
	File1.close();

	//this bellow, will return error, because I use the ios::noreplace
	//open_mode, which returns error if the file already exists.
	ofstream Test("file2.txt",ios::noreplace);

	//The error that the last line returned is ios::failbit, so let's show it
	if(Test.rdstate() == ios::failbit)
		cout << "Error...!\n";

	Test.clear(ios::goodbit); //set the current status to ios::goodbit

	if(Test.rdstate() == ios::goodbit) //check if it was set correctly
		cout << "Fine!\n";

	Test.clear(ios::eofbit); //set it to ios::eofbit. Useless.

	if(Test.rdstate() == ios::eofbit) //and check again if it is this flag indeed
		cout << "EOF!\n";

	Test.close();

}
```

Instead using flags, you can use a function that actually does the same- checks if specific flag is up. The functions were mentioned before, so I won't mention them again. If you are not sure how to use them, just check out the place of the tutorial, where I showed few ways to check if file opening was successful or not. There, I used the fail() function.

Dealing with Binary files

Although, the files with formatted text (all that I talked about so far) are very useful, sometimes you may need to work with unformatted files- binary files. They have the same look as your program itself, and it is much different from what comes after using the << and >> operators. The functions that give you the possibility to write/read unformatted files are get() and put(). To read a byte, you can use get() and to write a byte, use put(). You should remember of get()… I have already used it before.

You wonder why even using it, the file looks formatted? Well, it is because I used the << and >> operators, I suppose.
get() and put() both take one parameter- a char variable or character.
If you want to read/write whole blocks of data, then you can use the read() and write() functions. Their prototypes are:

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

For the read() function, buf should be an array of chars, where the read block of data will be put. For the write() function, buf is an array of chars, where is the data you want to save in the file. For the both functions, num is a number, that defines the amount of data (in symbols) to be read/written.

If you reach the end of the file, before you have read "num" symbols, you can see how many symbols were read by calling the function gcount(). This function returns the number of read symbols for the last unformatted input operation.
And before going to the code examples, I have to add, that if you want to open a file for binary read/write, you should pass ios::binary as an open mode.

Now, let me give you some code examples, so that you can see how stuff works.

Example 1: Using get() and put()

```
#include <fstream.h>

void main()
{
	fstream File("test_file.txt",ios::out | ios::in | ios::binary);

	char ch;
	ch='o';

	File.put(ch); //put the content of ch to the file

	File.seekg(ios::beg); //go to the beginning of the file

	File.get(ch); //read one character

	cout << ch << endl; //display it

	File.close();
}
```

Example 2: Using read() and write()

```
#include <fstream.h>
#include <string.h>

void main()
{
	fstream File("test_file.txt",ios::out | ios::in | ios::binary);

	char arr[13];
	strcpy(arr,"Hello World!"); //put Hello World! into the array

	File.write(arr,5); //put the first 5 symbols into the file- "Hello"

	File.seekg(ios::beg); //go to the beginning of the file

	static char read_array[10]; //I will put the read data, here

	File.read(read_array,3); //read the first 3 symbols- "Hel"

	cout << read_array << endl; //display them

	File.close();
}
```

Some useful functions

tellg() – Retunrs an int type, that shows the current position of the inside-pointer. This one works only when you read a file. Example:

```
#include <fstream.h>

void main()
{
	//if we have "Hello" in test_file.txt
	ifstream File("test_file.txt");

	char arr[10];

	File.read(arr,10);

	//this should return 5, as Hello is 5 characters long
	cout << File.tellg() << endl;

	File.close();
}
```

tellp() – The same as tellg() but used when we write in a file. To summarize: when we read a file, and we want to get the current position of the inside-pointer, we should use tellg(). When we write in a file, and we want to get the current position of the inside-pointer, we should use tellp(). I won't show a code example for tellp() as it works absolutely the same way as tellg().

seekp() – Remember seekg()? I used it, when I was reading a file, and I wanted to go to specified position. seekp() is the same, but it is used when you write in a file. For example, if I read a file, and I want to get 3 characters back from the current position, I should call FileHandle.seekg(-3). But if I write in a file, and for example, I want to overwrite the last 5 characters, I have to go back 5 characters. Then, I should use FileHandle.seekp(-5).

ignore() – Used when reading a file. If you want to ignore certain amount of characters, just use this function. In fact, you can use seekg() instead, but the ignore() function has one advantage- you can specify a delimiter rule, where the ignore() function will stop. The prototype is:

```
istream& ignore( int nCount, delimiter );
```

Where nCount is the amount of characters to be ignored and delimiter is what its name says. It can be EOF if you want to stop at the end of the file. This way, this function is the same as seekg(). But it can also be '\n' for example, which will stop on the first new line. Here is example:

```
#include <fstream.h>

void main()
{
	//if we have "Hello World" in test_file.txt
	ifstream File("test_file.txt");

	static char arr[10];

	//stop on the 6th symbol, if you don't meet "l"
	//in case you meet "l"- stop there
	File.ignore(6,'l');

	File.read(arr,10);

	cout << arr << endl; //it should display "lo World!"

	File.close();

}
```