

## C++ Standard Library

# Input/Output with files

C++ provides the following classes to perform output and input of characters to/from files:

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

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Let's see an example:

<pre>// basic file operations #include &lt;iostream&gt; #include &lt;fstream&gt; using namespace std;  int main () {     ofstream myfile;     myfile.open ("example.txt");     myfile &lt;&lt; "Writing this to a file.\n";     myfile.close();     return 0; }</pre>	<pre>[file example.txt] Writing this to a file</pre>
---	--

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

But let's go step by step:

## Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

Where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

<code>ios::in</code>	Open for input operations.
<code>ios::out</code>	Open for output operations.
<code>ios::binary</code>	Open in binary mode.
<code>ios::ate</code>	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
<code>ios::app</code>	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
<code>ios::trunc</code>	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in   ios::out</code>

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a `bool` value of `true` in the case that indeed the stream object is associated with an open file, or `false` otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

## Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`.

## Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

<pre>// writing on a text file #include &lt;iostream&gt; #include &lt;fstream&gt; using namespace std;  int main () {     ofstream myfile ("example.txt");     if (myfile.is_open())     {         myfile &lt;&lt; "This is a line.\n";         myfile &lt;&lt; "This is another line.\n";         myfile.close();     }     else cout &lt;&lt; "Unable to open file";     return 0; }</pre>	<pre>[file example.txt] This is a line. This is another line.</pre>
--	---

Data input from a file can also be performed in the same way that we did with `cin`:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while (! myfile.eof() )
        {
            getline (myfile,line);
            cout << line << endl;
        }
        myfile.close();
    }
    else cout << "Unable to open file";

    return 0;
}
```

```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called `eof()` that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed `myfile.eof()` becomes true (i.e., the end of the file has been reached).

## Checking state flags

In addition to `eof()`, which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

### **bad()**

Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

### **fail()**

Returns true in the same cases as `bad()`, but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

### **eof()**

Returns true if a file open for reading has reached the end.

### **good()**

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function `clear()`, which takes no parameters.

## get and put stream pointers

All i/o streams objects have, at least, one internal stream pointer:

`ifstream`, like `istream`, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.