Using C++ File Streams

David Kieras, EECS Dept., Univ. of Michigan
Revised for EECS 381, 9/20/2012

File streams are a lot like cin and cout

In Standard C++, you can do I/O to and from disk files very much like the ordinary console I/O streams `cin` and `cout`. The object `cin` is a global object in the class istream (input stream), and the global object `cout` is a member of the class ostream (output stream). File streams come in two flavors also: the class ifstream (input file stream) inherits from istream, and the class ofstream (output file stream) inherits from ostream. Thus all of the member functions and operators that you can apply to an istream or ostream object can also be applied to ifstream and ofstream objects. However, file streams have some additional member functions and internal information reflecting how they are connected to files on the disk. *This document assumes the document Basic C++ Stream I/O.*

This document is concerned only with the handiest form of disk file, called a text file – it contains a sequence of ASCII characters, and is normally read or written from the beginning to the end. The stream-like nature of this is obvious; a file stream is simply connected at one end to a disk file. In an input file stream, characters are moved from the disk into the stream, and your program takes them out from the other end. For output, your program puts characters into the stream, and the system takes them out of the other end and copies them onto the disk.

The major difference between file streams and the two console streams is when and how the stream objects are created. The two console streams are created and set up for you when your program is started. The objects `cin` and `cout` are global objects created outside your program. In contrast, you are responsible for creating and setting up your own file streams; this is fine, since of course, you want to control which files are used for what purpose in your program.

Basics of using file streams

Let's get a quick overview, and then get into some details.

First, you declare a file stream object for each file you need to simultaneously access. In this example, we will use one input file, and output file. But your program can have and be using as many files simultaneously as you wish. You just declare a stream object for each file:

```
#include <iostream>
#include <fstream>  // the class declarations for file stream objects
using namespace std;
...
int main ()
{
       ifstream my_input_file;    // an input file stream object
       ofstream my_output_file;   // an output file stream object
...
}
```

The above example code declares two objects, an input file stream object, and an output file stream object. Of course, they can be named whatever you wish, like any other C++ variable.

A disk file consists of a body of text on the disk, arranged in a way determined by your computer's Operating System (OS), which is responsible for keeping track of the information. If the file is deleted, moved, expanded, contracted, etc., the OS keeps track of exactly where it is on the disk and how much of it there is. The C/C++ facilities for working with disk files actually call OS subroutines to do the work.

So before you can use a disk file, you have to establish a relationship between your file stream object and the disk file. More exactly, you have to ask the OS to connect your stream to the file. Fortunately, this is easy: Just tell the stream object that you want to "open" the disk file and supply the name of the disk file as a C-string; the open member function negotiates with the OS to locate that file on the disk and establish the connection between that file and your stream object. Continuing the example:

```
ifstream my_input_file;    // an input file stream object
ofstream my_output_file;   // an output file stream object

my_input_file.open("input_data"); // open the file named "input_data"
my_output_file.open("output_data");     // open the file named "output_data"
```

Now the stream `my_input_file` is connected to the text file on disk named "input_data" and the stream `my_output_file` is connected to the text file on disk named "output_data".

Instead of creating and then opening the file streams in separate statements, you can use a constructor that takes the file name as an argument; after doing the normal initializations, the constructor completes the initialization by opening the named file. The above four statements would then condense down to two:

```
ifstream my_input_file("input_data");   // create and open
ofstream my_output_file("output_data");
```

In both ways of opening a file, you can specify the file path or file name either with a C-string array or literal (as the above examples do), or in C++11 with a std::string. For example:

```
string filename;
cin >> filename;
ifstream my_input_file(filename);
```

When opening files, especially input files, is it critical to test for whether the open operation succeeded. File stream errors are discussed in more detail below. But for now, here is one way of doing this test using a member function that returns true if the file was successfully opened:

```
if (my_input_file.is_open()) {
       // can continue, file opened correctly
       }
```

Now that the file streams are open, using them could not be simpler. We can read and write variable values from/to the streams using the stream input and output operators just like with `cin` and `cout`. For example, to read an integer and a double from the input file:

```
my_input_file >> int_var >> double_var;
```

To output to the file:

```
my_output_file << "The integer is " << int_var << endl;
```

The contents of the input file are processed just like you were typing them in via `cin`, and the output going into the file looks identical to what is written on your display with `cout`. You can read or write as much information from the file as is appropriate. Since ifstream and ofstream inherit from istream and ostream, your definitions of overloaded operators << or >> for ostream and istream will automatically work for file streams.

An easy way to prepare an input file is to use a text editor that creates plain ASCII text files, such as using the "save as text" option in a Windows or Mac word-processor. But by far the most convenient approach is to use the same text editor you use for writing your programs - in Unix, this is just vi or emacs. The IDE program editors all work in terms of text files. In MSVC or CW, simply create a new file, type your text content into it, and save it. The same editors work great for viewing the contents of an output file as well. Just be sure that the last character in the file is a whitespace character such as a newline - this avoids some odd end-of-file behaviors.

When your program is finished reading from or writing to a file, it is considered good programming practice to "close" the file. This is asking the OS to disconnect your program from the disk file, and save the final state of the file on disk. For an input file, the effect of closing is minor. For an output file, it can be vital to close the file promptly; the file system normally "buffers" information before actually recording it on the disk - this saves a lot of time. But until the buffer is "flushed" and all the information actually written to the disk, the file is in an incomplete state. Fortunately, closing is even easier than opening:

```
my_input_file.close();     // close the file associated with this stream
my_output_file.close();    // close the file associated with this stream
```

A couple of handy tips: If you want to read a file twice with the same stream object, read it through once until end-of-file occurs, clear the end-of-file state with clear(), then close the file, then reopen it, and start reading it again. Opening an input file always resets it to read starting at the beginning. You can read information that your program has just written into an output file by closing the output file stream, and then reopening that same disk file as an input stream.

Now, on to the complications. These concern policies for opening files, some additional handy member functions, and what can go wrong - which includes how you tell when you have read all the way through a file.

Policies for opening files

When opening a file, you supply a string the contains the file name, and the OS searches its file system for the file of that name. OS's typically store files in a complicated hierarchical file structure set up by the user. Where should the OS look for the file?

In all of programming environments used in this course, the OS will look first for a file of the specified name in the same directory as where your project executable is. In this course, this is where we will place all disk files. If you do not want to place the file in that directory, you have to tell the OS where to look for it by specifying a "path" to follow to find the file. These are the same strings of directory names that are used all the time in DOS or Unix command lines. To specify a path, you have to look up the rules for path specifications for your platform.

To keep things simple, in this course we will always be placing files in the same directory as the project, so the path specification can consist of just the file name.

If you open an input file, and the OS cannot find a file with that name, it is an **error**, described in more detail below. You must either ask the user to supply a correct file name or path, or terminate the program. No program can read data from a non-existent file!

However, the situation is different with an output file. If you open a file for output, using only the normal default specifications (as above) and the OS cannot find a file with that name, *the open function creates a new empty file with that name*. Why? Because decades of experience shows this is the most convenient and sensible policy! This is almost certainly what you want! This is why it is the default behavior.

But what if you open a file for output using only the normal default specifications (as above) and it already exists and the OS finds it? The most sensible and convenient policy has proven to be the following: *The existing file is deleted, and a new empty file is created with the same name*. Again, this is almost certainly what you want! This is why it is the default behavior.

What if you want something different? Consult a reference for other member functions and opening options that you can supply. Full flexibility is available, but it is idiomatic to use the defaults when they apply (which they usually do).

Handy member functions for character and line input

In addition to using the stream input operator >>, there are member functions that you can use to input single characters or long sequences of characters such as lines. These are actually inherited from the istream class, so they can be used with either files or cin. Here are the handiest three, shown both as prototypes and as an example call:

```
int get();
cin.get();
```
Reads the next character, skipping nothing, and returns it as an integer. If eof is encountered, the function returns the special value defined as the macro EOF. The need to test for EOF means that this form is relatively inconvenient for reading from a file.

```
istream& get(char&);
cin.get(char_variable);
```
Reads the next character, skipping nothing, into the supplied char variable (notice the reference parameter). The returned value is a reference to the istream object; this returned value can be used to test the stream state.

```
istream& getline(char * array, int n);
input_file.getline(buffer_char_array, buffer_length);
```
This function reads characters into the pointed-to character array, reading until it has either encountered a `'\n'` or has read *n-1* characters. It terminates the string of characters with `'\0'` so you get a valid C string regardless of how much was read. As long as the supplied *n* is less than or equal to the array length, you will not overflow the array. Conveniently, the `'\n'` character is removed from the stream and is not placed into the array, which is usually what you want in order to read and process the information in a series of lines. The returned value is a reference to the istream object, which can be used to test the stream state. If it fills the array without finding the '\n', the input operation fails in the same way as invalid input (see below) to let you know that a newline was not found within the size of the line you are trying to read.

If you want different behaviors from these, consult a reference for different forms of get and getline that allow different possible terminators and different treatments of them. Note that if you want to read a line into a `std::string`, there is a special function just for this purpose, declared in `<string>`:

```
istream& getline(istream&, std::string&);
```

Because the string automatically expands as needed, reading a line into a `std::string` with this function cannot overflow, and so is by far the best way to process a file (or `cin` input) a line at a time.

What can go wrong

A stream object has a set of internal bits as data members that "remember" the state of the stream. Normally the stream is in the **good** state if it has been opened successfully and has not had a problem. An error turns on one of the error bits, and then the stream is no longer **good**, and the bit stays on until it is cleared with the `clear()` member function. Once an error bit is turned on, any I/O operation on the stream does nothing. So it is only valid to read from or write to a stream that is in the good state. There are member functions for testing the stream state. The names of these stream states, member functions, and bits are not very well chosen, so read the following carefully. As you will see, **bad** is **not** the exact opposite of **good**, and there are multiple ways to **fail**, only one of which is really **bad**.

If you try to open a file and it fails to open (e.g. the filename is wrong), the stream object goes into a not-good state; if you want to try to open it again (e.g. with the right filename), you must first call the clear() member function to reset the error bits.

While both input and output operations can produce errors, input errors are far more common and more important. Thus the remainder of this document discusses only input error situations; but be aware that output errors exist. There are three ways an input operation can produce an error:

1. The hardware or system software has a malfunction - this is a "hard" I/O error that your program typically can do nothing about except try to muddle through (maybe it won't happen a second time) or quit. This type of error sets the **bad** bit in the stream object, and the stream is no longer **good**. Fortunately, these are rare (unless you have flakey hardware). In this course, we will not require testing for this possibility.

2. The input characters are not consistent with the input request, and as a result, the request can not be satisfied. This can be due to errors in the input data (e.g. typing errors), or incorrect assumptions in how the program is written. Such an error turns on the **fail** bit, and the stream is no longer **good**. This situation is called **invalid input** in this document. The document on Basic C++ Stream I/O discusses invalid input and how to handle it in detail, so this document will only describe the kinds of errors that appear with file streams.

3. In attempting to satisfy an input request, an end of file (**eof**) is encountered that prevents the request from being satisfied. The **eof** bit will get set, and the stream is no longer **good**. Hitting the end of file means no more characters are available from the source (usually a file, but there are platform-specific keystrokes that can be used to signal end-of-file from the keyboard). Important: **eof** is not set when the last character in the file has been read, but only when an attempt is made to read another character after the last one. The stream object will stay not-good until the bits are cleared with the clear() member function. Note that closing the file will not clear the stream state, so if you read a file to the end, and want to read it again with the same stream object, you should clear the stream state, then close and re-open the file.

Often, when end-of-file is encountered, both the **eof** bit and the **fail** bit will both get set, because the attempt to satisfy the input request failed because it could not find what was needed, and in the course of trying to find it, we ran off the end of the file. However, in certain cases the end-of-file serves as a delimiter for numbers or strings, and so it is possible to get an **eof** condition without a **fail** condition. In this course, we simplify matters by requiring that all input text files have a newline character at the end of every line, including the last line. This means that all lines terminate with a whitespace to serve as a delimiter for the last data item on a line, including the last data item on the last line.

Now, getting an **eof** is not really an "error" unless there is *supposed* to be more input - checking for an **eof** condition is a customary and standard way of determining that there is no more data to process. So there are two possible

interpretations of the **eof** condition: it is either **expected** as a normal part of processing (we've read all the input and so are ready to use it), or **unexpected**, meaning that some input is missing (there's supposed to be more!) and so there is an error. Unexpected **eof**s in file input are usually handled by informing the user and then terminating the program.

Important: Do not test the state of the **eof** bit (with the `eof()` member function) in order to control an input reading loop! This will not detect **fail** or **bad** conditions, and the end of file condition is not raised until you try to read past the input; the last successful input does not set the **eof** condition! Instead control the reading loop testing for the stream being in the **good** or **not-good** state. Use `eof()` only to determine why the stream is no longer good. See the example below.

Also important: Remember that once the stream is no longer **good**, it will stay that way, and any additional input operations will do **nothing**, no matter what they are or what is in the input. You have to **clear** the stream by resetting the error bits before input will work on the stream again. If you don't clear the stream state, your program will drop through all the remaining input statements doing nothing; often, your program will appear to hang, or loop forever. While this may seem cranky, it is a way to ensure that if your program gets incorrect input, and your code does not detect and handle it appropriately, something obviously wrong will probably happen.

Member functions for detecting and handling stream errors.

You can test the state of a stream using various member functions and also with a conversion operator that allows you to test the stream object itself. These are:

`bool good();`
: Returns true if none of the error bits are on and the stream is ready to use (is open). This is a good choice for asking "is everything OK?"

```
if (stream_object)
if (!stream_object)
if (stream_object >> var)
while (stream_object >> var)
```

: These tests of the whole stream object correspond to using `good()` — The stream classes have a *conversion operator* that converts the stream object to the same true-false value that is returned by the `good()` function. The first test is true if the stream is in a good state; the second if the stream is not in a good state. The third and fourth examples test the result of the input operation (remember that the result of the input operator is the stream object itself). The fourth example form is commonly used to repeatedly read the stream until an end of file condition.

`bool is_open();`
: Returns true if the stream is open. A good choice for testing for a successful opening because its name makes the purpose of the test more obvious.

`bool bad();`
: Returns true if the **bad** bit is on as a result of a "hard" I/O error. It is ***not*** the opposite of `good()`.

`bool fail();`
: Returns true if the **fail** bit is on due to invalid input or the **bad** bit is on (odd, but Standard) (see the Basic C++ Stream I/O handout).

`bool eof();`
: Returns true if the **eof** bit is on due to trying to read past the end of the file.

`clear();`
: Resets all of the error bits to off. Does not change which characters will be read next from the stream.

Recovering from an input error

The action your program takes on an input error depends on the type of error encountered. If it is invalid input, your program needs to clean up the input and clear the stream, and attempt to continue. If it is an expected eof, the program simply goes to the next step in the processing. But if the eof is unexpected, something is wrong, and the user needs to be informed. Finally, if you are checking for hard I/O errors, you need to deal with it if it turns out to be the problem.

A simple example

The following example program opens an input and output file, checks each one for being open, and then reads integers from the input file and writes twice the value of each one to the output file. It continues until the input stream is no longer good. This simple pattern would be justified if the programmer was confident that (1) the input data was always valid (no garbage); (2) No hard I/O errors would occur; (3) If 1 and 2 turn out to be false, we can recognize it by other means. Under these conditions, the stream input fails only at end of file, making for a very simple file-reading loop. For some kinds of data files (e.g. string data) where invalid input cannot happen, this approach is usually adequate. The pattern represented by this program is:

Attempt to read some input.
Check the stream state.
If the state is **good**, process the input.
If the state is **not good**, assume **expected eof** condition and continue processing.

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
	ifstream input_file("data.input");		// open the input file
	if (!input_file.is_open()) {	// check for successful opening
		cout << "Input file could not be opened! Terminating!" << endl;
		return 1;
		}
	ofstream output_file("data.output");	// open the output file
	if (!output_file.is_open()) { // check for successful opening
		cout << "Output file could not be opened! Terminating!" << endl;
		return 1;
		}
	// read as long as the stream is good - any problem, just quit.
	// output is each number times two on a line by itself
	int datum;
	while (input_file >> datum) {
		output_file << datum * 2 << endl;
		}
	input_file.close();
	output_file.close();
	cout << "Done!" << endl;
	return 0;
}
--- input file ---
12 34 23
34
6
89
--- output file ---
24
68
46
```

68
12
178

Example: Testing all the error bits with member functions.

This rather paranoid example has a get_int function that tries to read an integer from file, and checks everything. The function has three reference-type parameters: an input stream to read from, a bool good_flag parameter, which will be set to true if a valid integer was read in, and an integer variable, in which the read-in value will be stored. It returns a bool value that is true if we should continue reading from the stream, and false if we should stop. Function main() calls get_int in the condition of a while loop which continues as long as get_int returns true. When get_int returns, if value_is_good is true, then datum contains an integer that was successfully read in. We print it, and then continue to the loop. When get_int returns false, we stop looping and terminate.

Inside get_int, we first attempt to read an integer. If the stream is good, the variable contains a valid value, and we can continue reading the stream. If eof was encountered, a valid variable value was not obtained, and it time to stop reading the input. Likewise, if there was a "hard I/O" error. Finally, if the input was invalid, a valid value was not obtained. The example follows a policy to skip the rest of the line is skipped, and we should continue trying to read the stream (unless we hit end of file while skipping the rest of the line). The pattern represented by this example is:

Attempt to read some input.
Check the stream state.
If the state is **good**, process the input.
If the state is **not good**, do the following:
    If the state is **eof**, then handle an end-of-file condition:
        If an eof is expected and normal, continue processing.
        Otherwise, it is an error; print a message, and take appropriate action.
    If the state is **bad** bit; print a message and terminate.
    If the state is **fail** due to invalid input, do the following:
        Print a message informing the user of the bad input.
        Clear the stream state with the `clear()` function.
        Skip the offending material.
        Resume processing.

```
#include <iostream>
#include <fstream>
using namespace std;
bool get_int(istream&, bool&, int&);
int main ()
{
       int datum;
       bool value_is_good = false;
       ifstream input_file("data.input");      // open the input file
       if (!input_file.is_open()) {     // check for successful opening
              cout << "File could not be opened! Terminating!" << endl;
              return 1;
              }
       // continue reading integer values as long as get_int returns true
       // but don't use the value unless value_is_good is true
       while (get_int(input_file, value_is_good, datum))
              if (value_is_good)
                     cout << "value read is " << datum << endl;
       input_file.close();
       cout << "Done!" << endl;
       return 0;
}
```

```
bool get_int(istream& in_strm, bool& good_flag, int& x)
{
	bool continue_flag;

	in_strm >> x;
	if (in_strm.good()) {
		good_flag = true;
		continue_flag = true;	// can keep going
		}
	else if (in_strm.eof()) {
		cout << "End of file encountered." << endl;
		good_flag = false;	// input value was not obtained
		continue_flag = false;	// time to stop
		}
	else if (in_strm.bad()) {
		cout << "Hard I/O error" << endl;
		good_flag = false;
		continue_flag = false;	// give up!
		}
	else if (in_strm.fail()) {
		cout << "Invalid input - skipping rest of line" << endl;
		in_strm.clear();	// don't forget! Must clear the stream to read it!
		char c;
		while (in_strm.get(c) && c != '\n'); // may hit eof while skipping
		good_flag = false;	// value is not good
		if (in_strm.good())	// did we hit eof or something else?
			continue_flag = true;	//  no - can keep going
		else {
			continue_flag = false; // yes - time to stop
			cout << "End of file or error while skipping rest of line." <<
endl;
			}
		}
	else {
		cout << "Should be impossible to be here!" << endl; // for demo only!
		good_flag = false;
		continue_flag = false;
		}
	return continue_flag;
}

--- input file ---
12
34.5
6t7
89
x
--- output ---
value read is 12
value read is 34
Invalid input - skipping rest of line
value read is 6
Invalid input - skipping rest of line
value read is 89
Invalid input - skipping rest of line
End of file or error while skipping rest of line.
Done!
```