Basics of I/O Streams and File I/O

This is like a cheat sheet for file I/O in C++. It summarizes the steps you must take to do basic I/O to and from files, with only a tiny bit of explanation. It is not a replacement for reading a thorough explanation of file streams in C++.

- Streams: A **stream** is a flow of characters.
- An input **stream** is a flow of characters into the program.
- An output **stream** is a flow of characters out of the program.
- `cin` is a predefined input stream (defined in `<iostream>`).
- `cout` is a predefined output stream (defined in `<iostream>`).
- `cerr` is a predefined output error stream (defined in `<iostream>`), to which you should send all error messages. It defaults to the terminal (just like `cout.`)
- Streams are defined in the streams library header file, `<iostream>` and implemented in the `iostream` library.
- What if you want characters to come from a file?
  - You need an ***input file stream.***
- What if you want your output characters to go into a file?
  - You need an ***output file stream.***
- Both are called ***file streams.***
- File streams are like `cin` and `cout`, except that the flow is to and from a ***file***. They share many of the same properties and functions.
- File streams are a special kind of I/O stream. C++ defines file streams in a library called `fstream`, whose header file is `<fstream>.`
- How can tell when there are no more characters left to read from the input file associated with a particular file stream?
  - You need to detect the "end of file" condition. There are a few ways to do this. Below I show you the easy way.

How to use a file named `infile.txt` as input and a file named `outfile.txt` as output:

1. Include the `fstream` library header file in your program and "open" the `std` namespace:

```
#include <fstream>
```

```
using namespace std;
```

2. Declare an input file stream with whatever name you choose in your main program (I like `fin`) and an output file stream (`fout` here) as follows:

```
int main()
{
     ifstream  fin;
     ofstream  fout;
...
```

3. Before your program attempts to read any input or write any output, open the file streams and associate them with the actual files:

```
fin.open("infile.txt");
fout.open("outfile.txt");
```

   Note that this won't work if the files are not in the same directory as the running program. In that case you have to specify the absolute or relative path name.

4. Instead of using statements with `cin` for input and `cout` for output, use `fin` in place of `cin`, and use `fout` in place of `cout`:

```
int first, second;
fin >> first >> second;
fout << "the sum is " << first + second;
```

5. Before your program's closing **`return`** statement, close all open files:

```
fin.close();
fout.close();
```

   `s.close()` closes a file. This must be done or else the chars that were written to the output may be lost. Also, you can reuse a stream to open a second file by closing and reopening with the new file.

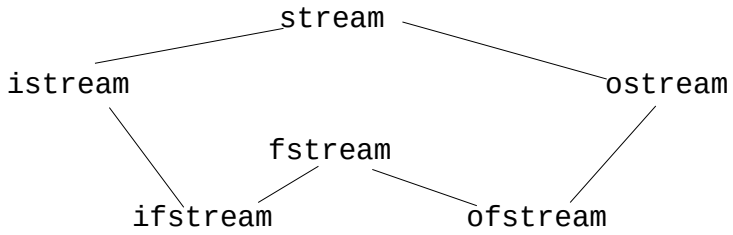
   Note too that these are void functions with no arguments.

Example

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

int main()
{
```

```
      ifstream  fin;   // declares an object of type ifstream
      ofstream fout;   // declares an object of type ofstream
                       // ifstream and ofstream are defined in
<fstream>

      fin.open("infile.txt");
      fout.open(“outfile.txt");

      // for any stream object s, s.open(<filename>) opens filename
      // and connects the stream s to it so that chars can flow between
      // them.

      int num1, num2, num3;
      fin >> num1 >> num2 >> num3;
      fout << "The sum is " << num1+num2+num3 << endl;

      fin.close();
      fout.close();
      return 1;
}
```

- `istream` is a linear connection from a program to a source of characters
- `ostream` is a linear connection from a program to a repository of characters.
- `fstreams` have all of the properties and functions of `streams`, and extra functions and properties of files too. The inheritance relationship among some of these classes is depicted below.



Practical issues with file streams.

1. ***You should never assume that opening a file was successful. You must always test that it succeeded before continuing***. Testing if a file stream function call was successful:

```
in_stream.open("mydata");
if ( in_stream.fail() )
{
      cerr << "Could not open mydata.\n";
      exit(1);  // 1 indicates an error occurred
}
```

You need to include `<cstdlib>` to use the `exit()` call.

2. The `fail()` function works on any stream.
3. You do not need to provide prompts for input when reading from files, and you should not!
4. ***If you want the user to be prompted to enter a file name, you can use something like this:***

```
char  filename[512];
cout << "Enter the absolute path to the file:";
cin  >> filename;
fin.open(filename);
if ( fin.fail() ) {
    cerr << "Could not open file " << filename << endl;
    exit(1);
}
```

In the preceding code snippet, if you have never seen a C string, then the first line will look strange to you. It simply declares an object that can store up to 511 characters. The 512th character is reserved to store a `NULL` character, ASCII '`\0`'. When the `istream` extractor `>>` is given such an object and the user enters text, the extractor automatically inserts the `NULL` character at the end of the text.

You cannot give `fin.open()` an argument that is a C++ string; you must give it a C string.

Detecting when there is no more data in the input file stream

If your program tries to read more data from any input stream, but there is none, the extraction operator will return the `boolean` value `false`. When you used `cin` as the input stream, you may not have thought about what it means to run out of data, but with files this happens because files are of finite size.

If `fin` is an input file stream, and `str` is a variable of type `string`, then

```
fin >> str;
```

will have the value `false` if when the instruction is executed, there are no characters left to be read in the stream. This means that you can write a loop like

```
while ( fin >> str ) {
    cout << str;
}
```

and it will keep looping until the file associated with `fin` has been read completely. Remember that the extraction operator skips over whitespace. If there is whitespace after the last string, it will attempt to look for the next non-whitespace sequence and then it will return false.

Even with `cin`, there can be a synthetically generated end-of-file condition. In UNIX, the user can type Control-D on the keyboard, and that will usually cause an end-of-file signal to be sent to your program. (It may not if the user has configured his or her shell to turn off that notification.) In Windows, Control-Z has the same effect.

How do you detect the end-of-file when you are not using the extraction operator?

There are other ways to read from a file, such as the `get()` member functions and the `getline()` global function.

`get()` reads a character at a time. It does not skip whitespace and it does not skip newline characters. It reads each and every character, advancing the stream pointer to the next character in the stream. If `instream` is any open input stream and `ch` is a character variable,

```
instream.get(ch);
```

will put the next character in the stream into `ch`. If there is no character left in the stream, then `instream.eof()` will become true. But it does not become true until after the attempt is made to read from the stream. Therefore the correct way to read characters from a stream using this form of `get()` is as follows:

```
char ch;
instream.get(ch);
while (  !instream.eof() ) {
     cout.put(ch);
     instream.get(ch);
}
```

It is necessary to first call `get()` before checking the return value of `eof()` and that is why the call to `get()` appears before the loop begins and also at the end of the loop. There is another form of `get()` that returns the value of the character as an integer and that has no arguments. It can be used as follows:

```
char ch;
while ( (ch = instream.get()) && !instream.eof() )
     cout.put(ch);
```

In this loop the character is read in the while loop condition itself. Because the return value will always be a non-zero, this evaluates to true and the `eof()` function is checked in each iteration.

How can you append to an existing file?

Appending to a file is easy. When you use the open() member function to open an output file stream, you add an extra argument:

```
fout.open("outfile", ios::app);
```

The argument ios::app tells the compiler that the file should be appended to and not overwritten.