Intro to File Input/Output in C++

Redirection

One way to get input into a program or to display output from a program is to use *standard input* and *standard output*, respectively. All that means is that to read in data, we use `cin` (or a few other functions) and to write out data, we use `cout`.

When we need to take input from a file (instead of having the user type data at the keyboard) we can use input redirection:

```
% a.out < inputfile
```

This allows us to use the same `cin` calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., `inputfile` above) instead of the keyboard.

Similarly, there is *output redirection*:

```
% a.out > outputfile
```

that allows us to use `cout` as before, but that causes the output of the program to go to a file (e.g., `outputfile` above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time...

```
% a.out < inputfile > outputfile
```

C++ File I/O

While redirection is very useful, it is really part of the operating system (not C++). In fact, C++ has a general mechanism for reading and writing files, which is more flexible than redirection alone.

iostream.h and fstream.h

There are types and functions in the library **iostream.h** that are used for standard I/O. **fstream.h** includes the definitions for stream classes **ifstream** (for input from a file), **ofstream** (for output to a file) and **fstream** (for input to and output from a file). Make sure you always include that header when you use files.

Type

For files you want to read or write, you need a file stream object, e.g.:

```
ifstream inFile;  // object for reading from a file
ofstream outFile; // object for writing to a file
```

Functions

Reading from or writing to a file in C++ requires 3 basic steps:

1. Open the file.
2. Do all the reading or writing.
3. Close the file.

Following are described the functions needed to accomplish each step.

1. **Opening a file:**

   In order to open a file, use the member function `open()`. Use it as:

   ```
   inFile.open(filename, mode);
   outFile.open(filename, mode);
   ```

   where:

   - *filename* is a string that holds the name of the file on disk (including a *path* like `/cs/course` if necessary).
   - *mode* is a string representing how you want to open the file. Most often you'll open a file for reading (`ios::in`) or writing (`ios::out or ios::app`).

   Note that `open()` initializes the file object that can then be used to access the file. After opening the file, we should test the file object (e.g., with `!` below) to make sure it was properly opened (e.g., an open may fail if we don't have the correct permissions or the file doesn't exist when opening for reading).

   Here are examples of opening files:

   ```
   ifstream inFile;
   ofstream outFile;
   char inputFilename[] = "in.list";
   char outputFilename[] = "out.list";


   inFile.open(inputFilename, ios::in);

   if (!inFile) {
     cerr << "Can't open input file " << inputFilename << endl;
     exit(1);
   }

   outFile.open(outputFilename, ios::out);

   if (!outFile) {
     cerr << "Can't open output file " << outputFilename << endl;
     exit(1);
   }
   ```

   Note that the input file that we are opening for reading (`ios::in`) must already exist. In contrast, the

output file we are opening for writing (`ios::out`) does not have to exist. If it does not, it will be created. If this output file does already exist, its previous contents will be thrown away (and will be lost).

Note: There are other modes you can use when opening a file, such as append (`ios::app`) to append something to the end of a file without losing its contents...or modes that allow you to both read and write.

---

2. **Reading from or writing to a file:**

   Once a file has been successfully opened, you can read from it in the same way as you would read with `cin` or write to it in the same way as you write using `cout`.

   Continuing our example from above, suppose the input file consists of lines with a *username* and an *integer test score*, e.g.:

```
in.list
-------
foo 70
bar 98
...
```

   and that each username is no more than 8 characters long.

   We might use the files we opened above by copying each username and score from the input file to the output file. In the process, we'll increase each score by 10 points for the output file:

```
char username[9];  // One extra for null char.
int score;

...

while (!inFile.eof()) {
  inFile >> username >> score;
  outFile << username << " " << score+10 << endl;
}
```

   In the `while` loop, we keep on reading `username`and `score` until we hit the end of the file. This is tested by calling the member function `eof()`.

   The bad thing about using `eof()` is that if the file is not in the right format (e.g., a letter is found when a number is expected):

```
in.list
-------
foo 70
bar 98
biz A+
...
```

   then `>>` will not be able to read that line (since there is no integer to read) and it won't advance to the next line in the file. For this error, `eof()` will not return true (it's not at the end of the file)....

Errors like that will at least mess up how the rest of the file is read. In some cases, they will cause an *infinite loop*.

One solution is to test against the number of values we expect to be read by >> operator each time. Since there are two types *a string* and *an integer*, we expect it to read in 2 values, so our condition could be:

```
while (inFile >> username >> score) {
...
}
```

Now, if we get 2 values, the loop continues. If we don't get 2 values, either because we are at the end of the file or some other problem occurred (e.g., it sees a letter when it is trying to read in a number), then the loop will end (>> will return a 0 in this case).

Note: When you use `eof()`, it will not detect the end of the file until it tries to read past it. In other words, they won't report end-of-file on the last valid read, only on the one after it.

---

3. **Closing a file:**

   When done with a file, it must be closed using the member function `close()`.

   To finish our example, we'd want to close our input and output files:

   ```
   inFile.close();
   outFile.close();
   ```

   Closing a file is very important, especially with output files. The reason is that output is often *buffered*. This means that when you tell C++ to write something out, e.g.,

   ```
   outFile << "Whatever!";
   ```

   it doesn't necessary get written to disk right away, but may end up in a *buffer* in memory. This output buffer would hold the text temporarily:

   ```
   Sample output buffer:
   ---------------------------------------------
   | a | b  | c | W | h | a | t | e | v | e | r |
   ---------------------------------------------
   | ! |    |   |   |   |   |   |   |   |   |   |
   ---------------------------------------------
   |   |    |   |   |   |   |   |   |   |   |   |
   ---------------------------------------------
   |   |    |   |   |   |   |   |   |   |   |   |
   ---------------------------------------------
   ...
   ```

   (The buffer is really just 1-dimensional despite this drawing.)

   When the buffer fills up (or when the file is *closed*), the data is finally written to disk.

   So, if you forget to close an output file then whatever is still in the buffer may not be written out.

Note: There are other kinds of buffering than the one we describe here.

A complete program that includes the code above, plus input files to use with that program, is available to download.

Special Files

There are 3 special file objects that are always defined for a program. They are `cin` (*standard input*), `cout` (*standard output*) and `cerr` (*standard error*).

Standard Input

Standard input is where things come from when you use `cin`. For example,

```
cin >> val;
```

Standard Output

Similarly, *standard output* is exactly where things go when you use cout. For example,

```
cout << "Value = " << val << endl;
```

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless *redirection* is used.

Standard Error

Standard error is where you should display error messages. We've already done that above:

```
cerr << "Can't open input file in.list!" << endl;
```

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

```
% a.out > outfile
```

only redirects stuff going to standard output to the file **outfile**... anything written to standard error goes to the screen.

BU CAS CS - Intro to File Input/Output in C++
This page created by Saratendu Sethi <sethi@cs.bu.edu>.
Material adapted for C++ from Intro to File Input/Output in C.