CSCI 262 Data Structures

Spring 2019

Lab 2: I/O

(Back to Labs)

Goals

- Learn how to read simple values from the console or a file
- Learn how to read lines from the console or a file
- Learn how to use string streams for simple string parsing

Overview

This lab will walk you through some exercises designed to reinforce your understanding of streams (chapter 8 of your textbook) as well as teach you how to use stringstreams (section 8.4 of your textbook) and to help prepare you for project 1. As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

Instructions

Task 0: Setup. Go ahead and create a CLion project for this lab, by choosing "New Project..." from the File menu. Note the dialog you get after doing this gives you the option to specify where to put your new project - you might choose "Z:\csci262\lab02" or something similar. Note that CLion gives you a "Hello, World!" program to get you started (how nice!).

Also, download lab02.zip, which contains some text files you will use later on. Unzip or open the .zip archive file, and move or copy the three text files to the "cmake-build-debug" directory under the new directory you just created. (You might have to wait a few seconds for CLion to create the directory for you.) This is where CLion will create and run your executable program by default, which means it is also where your program will look for files when you try to open them using an ifstream.

Task 1: create an "echo" program

1. For this task, you can work in the main.cpp that CLion has already created for you. Overwrite the code in main() with this code:

```
string x = "";
while (x != "exit") {
    cout << "Enter something: ";
```

```
        cin >> x;
        cout << "You entered: \"" << x << '\"' << endl;
    }
```

Note you will need `#include <iostream>`, and `using namespace std;` at the top of your file as well.

2. Go ahead and run the code in CLion using the green triangle button in the upper right corner. You should get a window pane at the bottom of CLion where you can interact with the running program. Experiment with entering various words, numbers, **and sentences** in response to the prompt from the program. Does the program work the way you expected it to? Type "exit" to quit the program, then run it again and enter "one two three" at the prompt.

   The behavior you see is due to the rather simple-minded way the >> operator works. By default (there are ways to override various bits of this), >> skips over any whitespace in the input *buffer* (a bit of memory which holds on to everything that has been typed at the console but not yet consumed by the program) and then reads in the next contiguous chunk of non-whitespace characters. The >> operator attempts to interpret what it reads as a value of the type corresponding to the variable on the right-hand side of >> - in this case, anything you type in is ok, since we are trying to read values into a string variable, which can hold any characters.

   When we enter three strings all at the same time, however, our loop executes three times without waiting for additional input; this is because the >> operator consumes only the characters up to the next whitespace, then stops. The remaining characters (including the whitespace) are still in the input buffer.

   Our string has two spaces in between our three words, leading to our loop running three times.

3. Let's try something else. Change the variable `x` to be an `int`, and change the `if` statement to stop when the value `-1` is entered. Experiment again with entering values, starting with integers only. What happens when you enter a non-integer? **If you need to, you can kill the currently running program in the console by clicking the red square button (either in the upper right corner of CLion, or just to the left of the console pane).**

   Can you guess why this is happening? The >> is doing its best to interpret what you typed in as an integer, but when it fails, it simply reads in a zero. However, everything you entered (which couldn't be interpreted as an integer) is still in the input buffer! Worse, the stream at this point has entered a *bad* state, and won't let you read anything more until you clear the error. A better program would check, after each >> operation to see if `cin.good()` returns true; if not, the best thing to do would be to clear the error (`cin.clear()`) and then ignore the rest of the line (`cin.ignore(numeric_limits::max(), '\n')`).

   In general, it can be difficult to do what we really want to with the >> operator when doing console I/O (Input/Output).

4. When working with console I/O in particular, it is usually better to work with the free function

`getline()`, which lives not in `iostream` but in `string`. (The documentation on this is hard to track down, so here's a link: http://www.cplusplus.com/reference/string/string/getline/.)

Modify your code to look like the following:

```
string x = "";
while (x != "exit") {
    cout << "Enter something: ";
    getline(cin, x);
    cout << "You entered: \"" << x << '\"' << endl;
}
```

Experiment with this program. The function `getline()` works better with the console simply because it takes in an entire line of input from the user at one time, discarding the line ending whitespace (the line ending sequence varies from system to system; on Windows it is two characters, `"\r\n"`, while on linux it is just `"\n"`.) Of course, if what you really want is some `int` value or some sub-component of the input line, you have more work to do; in this case, you may wish to use an `istringstream` object to further break down the string (we'll discuss istringstreams in section 3 below).

Task 2: copy a file

1. The objects `cin` and `cout` are special objects that connect to underlying mechanisms in the C++ runtime code for managing console input and output. (These are called the standard input and output streams - you might see programmers referring to "stdin" or "stdout".) The notion of a *stream* is an *abstraction* of the essential properties of data devices such as the console, files, network connections, and much more.

To work with files in C++, we use file abstractions from the <fstream> library, which includes the `ifstream` and `ofstream` classes. The `ifstream` class allows for input from files (basically `cin`, but with files), and the `ofstream` allows for output to files (basically `cout`, but with files). (It is also possible to work with a lower level class that lets you both read and write the same file - not a beginner exercise!)

Go ahead and `#include <fstream>` in your program. To open a file for reading (input), you have a couple of choices. You can create an ifstream object named `fin` unbound from any file using

```
ifstream fin;
```

To subsequently open a file with this ifstream object, use the `open` method:

```
fin.open("haiku.txt");
```

Alternately, you can use the ifstream constructor to open the file and create the ifstream object all at the same time:

```
ifstream fin("haiku.txt");
```

Choose one of these methods and create code (you can discard your old code, or make a new file) to open the file "haiku.txt". The file "haiku.txt" will need to be in the same directory that you run your compiled program from.

2. Once you have your ifstream object, you can do with it more or less what you could do with `cin`. One difference is that console input is effectively infinite - if you try to input from the console, the program will wait until someone types something. Files, on the other hand, end at some point. Fortunately, it is easy to detect when you have reached the end of file (or EOF): use the `eof()` method. Write a simple loop to read in each line of the haiku file and print it to the console:

```
while (!fin.eof()) {
    ...
}
```

3. All files should be closed when they are no longer needed, using the `close()` method of ifstream. (All files are automatically closed when the program closes, but having files open when not needed can lead to various kinds of problems, so get in the habit of always writing the close statement to go with every open.)

4. Opening files for writing (output) is pretty much identical to opening files for reading, except that you need an `ofstream` object instead of `ifstream`. **Important:** when you open a file with an `ofstream` the default behavior either a) creates the file if it didn't exist before or b) truncates the file if it already exists. Once you have an ofstream object, you can use it identically to how you use `cout`, but anything you output goes into the file. As with input file streams, output file streams should be closed when you are done with them!

   With this knowledge, you are ready to make a program to copy a text file. Modify your code to open a new file named "haiku2.txt" for output, and write the lines from "haiku.txt" to "haiku2.txt" instead of to the console. As a final step, you could modify your program to prompt the user for the file to copy, and the file to save the copy to, rather than hardcoding the filenames.

Task 3: Parse a string

Parsing in the context of computers usually means taking a string apart and identifying the parts. There are various ways to do this, including using the standard *regular expression* library (`<regex>`), or even a full fledged parser generator program like `bison`. Today you're going to do something much simpler, which is to use the *stringstream* library to treat strings as if they were files. Stringstreams can be confusing, so we'll overexplain them a bit. Effectively, a stringstream object created from a string would function the same way as an ifstream object created from a file with the same contents as that string. This allows you to use a string as a stream (see what they did there with the name?) and parse out individual whitespace separated parts of the string.

If you're still having trouble understanding what we're talking about, imagine a string. Now imagine a file that has the same contents as that string (e.g. `string a = "hello"` and a text file with only one line that says

"hello"). Creating an ifstream from said file would allow you to use the file as a stream, and your ifstream object would funcion like `cin`; creating a stringstream object from said string would create a stream object that would behave the same way as our `ifstream` object since the file and the string had the same contents. Let's use stringstreams now!

1. Again start a new program. This time, you need to `#include <sstream>` as well as `<iostream>`.

2. Next, write code to prompt the user to enter a string, and use `getline()` to obtain the string from the user.

3. Now, create an `istringstream` object on the string. This is easily done using the `istringstream` constructor. For instance, if your string is held in the variable `str`, and you want to create a stringstream object named `sin`, you can create the object using

   ```
   istringstream sin(str);
   ```

   Here, I've named the object `sin` ("string input"), but you can name it whatever you want.

4. Remember how we used the `eof()` method to detect when we hit the end of a file? The same method is used for istringstreams to detect when you reach the end of the input string. So you can now write a loop that uses the >> operator on your new istringstream object to get the individual strings from the string passed in by the user.

5. For bonus points (well, not really as this part isn't graded), try using the >> operator, the `good()` method, and the `clear()` method to take in an arbitrary string and parse it into doubles (when possible) and strings for anything not matching a double. (It would be nice to further distinguish between integers and doubles, but that takes a bit more work.) Your program should output what kind of value you read, and the value itself. For instance, if the input is "The fourth digit of 3.14159265 is 1", your output might look like

   ```
   string: The
   string: fourth
   string: digit
   string: of
   number: 3.14159265
   string: is
   number: 1
   ```

Task 4: Put it all together

For the final task, you are provided with a text file, "lines.txt", and asked to discover two facts from the file. The file contains a large number of English words (from a Scrabble dictionary); one of these is larger than all of the others. Your job is to a) find that word and tell us what it is and b) tell us what line number the word is on. The file is laid out as follows:

- The first line has three integers. The first integer is the number of lines after the first line. The second

and third integers are the minimum and maximum number of words on each line, respectively.
- Each subsequent line contains words separated by spaces.

Assume that the first line of text (not the first line of the file) is line number 1 for line numbering purposes. To find the information we're asking for, you'll want to do something like the following:

1. Read in the first line of the file to get the number of lines of text.
2. Read in the lines of text one by one using `getline()`.
3. Read in each word from the line using an `istringstream` object.
4. Keep track of the longest word you've found, and what line you found it on.

Finish up: submit your answers to the Lab 2 quiz on Canvas.