Lab: I/O, Control Flow, and Strings

May 25, 2013

Contents

In this lab we are going to explore some of the interactions among our three most-recent major topics - I/O, control flow, and strings. In particular, we are going to explore one of the most common I/O tasks, reading an entire file of data from beginning to end.

Get into Code::Blocks and create a new project for this lab.

1 Checking the Status of Input

When we are reading repeatedly with an input stream, we have certain tests that we can apply to determine how well the input in progressing. These include:

- `eof()`: Have we reached the end of the input?
- `fail()`: Did the prior input fail for some reason? For example, if we tried to read an `int` and ran into a bunch of alphabetic characters in the input, then `bad()` would become true. When an input has failed, no characters have been lost and it is, theoretically, possible to continue reading if you can figure out what went wrong.
- `bad()`: Did the prior input fail for some reason that makes the stream unusable? For example, if we tried to read something from a file that does not exist or after we had already reached the end of file, the stream would then be `bad()`. No recovery is possible.
- `good()`: None of the above are true. If `good()` is true, we know that our previous input operation succeeded. But if `good()` is false, any subsequent input requests will be ignored, so we know that any subsequent I/O will not work.

Create the following program.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   ifstream input ("data1.txt");
```

```
    for (int i = 0; i < 5; ++i)
    {
        int k = -12345;
        input >> k;
        cout << "k: " << k
            << "\teof: " << input.eof()
            << "\tfail: " << input.fail()
            << "\tbad: " << input.bad()
            << "\tgood: " << input.good()
            << endl;
    }
    return 0;
}
```

Then create a data file `data1.txt` in the same directory as your code (easiest way to do this is to use Code::Blocks - go to the File menu, select "New", and then "Empty File".) It should contain

```
1
2 3
x
4 ab
5
```

Compile and run the program.

- Note the places where input fails. What flags get set when we hit the alphabetic characters?
- Note what happens to `k` when the input fails.

The input stream will not recover from an input error like this automatically. We will need to help it along. In particular, after a `fail`, no characters have been lost, including the alphabetic characters causing us the problems. So we might consider adding some code to try and recover from a failure by swallowing the next character. Change the program to:

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main()
{
    ifstream input ("data1.txt");
    for (int i = 0; i < 5; ++i)
    {
        int k = -12345;
        input >> k;
        cout << "k: " << k
            << "\teof: " << input.eof()
            << "\tfail: " << input.fail()
            << "\tbad: " << input.bad()
            << "\tgood: " << input.good()
            << endl;
        if (input.fail())
        {
            char garbage = '?';
            input.clear();
            input >> garbage;
            cout << "Attempting failure recovery, bypassing "
                << garbage << endl;
        }
    }
    return 0;
}
```

Compile and run. Notice that, after one failure, we do indeed succeed in moving on to the next number. The `clear()` call resets the eof, bad, and fail status of the input stream, forcing the stream to be `good()` again. This is important, because if the stream is not good, then our attempt to read and discard the `garbage` character would be ignored.

OK, let's try to push a little further through our input file. Change the for loop limit from "5" to "10". Compile and run again.

- Do we succeed in reading the last number (5) from the input?
  - Note that, to do so, we would have to bypass two garbage characters, not just one.
- Note when the `eof` flag becomes true. Is this, perhaps, one step later than you might have expected? We'll explore the reason for this, and the reason why `eof` and `fail` seem to be set at the same time, on the next page.

There's an odd little shortcut that C++ permits when running tests on the status of an input stream. If we use the stream in a context where a `bool` value is required (e.g., in the condition of an `if` statement), then the stream is "converted" to a `bool` by an automatic call to `good()`.

So, instead of writing

```
if (input.fail())
```

many C++ programmers would write

```
if (!input)
```

i.e., "if `input` is not good". Now, saying that an input stream is "not good" is not quite the same as saying that it has "failed". It might be not good because it is "bad" or because it is at end of file. But, in many cases, this works well enough, if not better.

A further shortcut is possible because >> is actually an operator in C++, like + or `/`, that returns a value. And that value is ... the stream that we are reading from! Now, when we write

```
input >> k;
```

we are telling C++ to "evaluate the expression "`input >> k;`" and throw away the result. That's perfectly legal in C++. For example, statements like this:

```
1 + 1;
3 * 2;
```

will compile and execute. They just don't do anything useful unless we save the results:

```
int two = 1 + 1;
int six = 3 * 2;
```

By contrast, `input >> k;` does something useful all by itself. But we *could* save the result if we wanted to:

```
istream& input2 = input >> k;
```

I don't know why we would want to, though, since it just gives us another name for the same input stream.

Much more useful is to use this expression result together with the automatic conversion to boolean. Change the program to

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream input ("data1.txt");
    for (int i = 0; i < 10; ++i)
    {
        int k = -12345;
        if (input >> k)
        { // input into k succeeded
            cout << "k: " << k;
        }
        else
        { // input into k failed
            char garbage = '?';
            input.clear();
            input >> garbage;
            cout << "Bypassing " << garbage;
        }
        cout << "\teof: " << input.eof()
            << "\tfail: " << input.fail()
            << "\tbad: " << input.bad()
            << "\tgood: " << input.good()
            << endl;
    }
    return 0;
}
```

The `if` statement in this program can be thought of as both performing an input and then immediately checking to see if that input succeeded. That's because, after the >> operator actually does the input, it returns the input stream, which is then converted to `bool` by implicitly calling `input.good()`.

Compile and execute this code. Notice that the stream stays good until we finally run out of input.

2 Free-Form Inputs

2.1 Numbers

Suppose that you have been asked to read a whole file of integers and told that they will appear in the file one per line. Try the following code:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream input ("data2.txt");
    while (!input.eof())
    {
        int k = -12345;
        input >> k;
        cout << k << endl;
    }
    return 0;
}
```

which certainly *looks* as if it should do the job. Create a data file `data2.txt` with the following content:

```
1
2
3
4
5
6
7
8
```

Compile and run the program.

OK, it *almost* worked. We'll look at what went wrong at then shortly. But, first, change the file `data2.txt` to this:

```
1 2
 3  4
5   6
 7    8
```

Run the program again. Did the output change?

Although our program is capable of reading numbers presented one per line, it's actually capable of reading any arrangement of numbers among the lines. Now, if we are *required* to enforce the input format, that would be a bad thing. But in many cases, guidelines like "one per line" are either guidelines for the people preparing the input or simply a description of the way that already existing files appear. If our input mechanism is more flexible than it needs to be, that's OK. (We'll look later at what to do when the lines really are important to us.)

It's also pretty clear from the way the numbers are arranged that the number of spaces between numbers is irrelevant. *Each call to » begins by discarding any whitespace characters at our current place in the input.* A *whitespace* character is one that prints as simply open space of some kind. The obvious whitespace characters are the blank `' '` and tab `'\t'` characters. But there's more going on here than meets the eye. If we were to "dump" our data file with a program that shows each and every character, even whitespace ones, we would find that the file actually looks like:

1		2	\r	\n		3			4	\r	\n	5				6	\r	\n		7					8	\r	\n

or, grouping by lines:

1		2	\r	\n				
	3			4	\r	\n		
5				6	\r	\n		
	7					8	\r	\n

The `\r` is the C++ representation of a "carriage return" character (ASCII code 13, also what you get by typing Ctrl-m) and `\n` designates a "line feed" character (ASCII code 10, also Ctrl-j).

- In Windows, these two characters are used to mark the end of a line.
- In Unix and Linux, line endings are marked with only a single character, a line feed. That's why we conventionally can end lines by writing strings with "\n":

```
cout << "One line\nAnother line\n";
```

On a Windows machine, the I/O system has to inject a \r every time we write a \n. On Unix/Linus systems, we really do just write the single line feed character. This difference also explains why you have to be a bit careful transferring even "plain text" files from one operating system to another. Windows programs think that text generated by Unix programs lack appropriate line terminators. Unix programs often think that files generated on Windows have an extra odd data character (Ctrl-m) at the end of each line.

Now we are in a position to discuss the odd little error that we observed at the end of our last program's execution. Suppose we start the program running and present it with the input:

1		2	\r	\n		3			4	\r	\n	5				6	\r	\n		7					8	\r	\n

Now, when we first hit the loop, `eof()` is clearly false, so we read and print an int (1).

The input remaining is

	2	\r	\n		3			4	\r	\n	5				6	\r	\n		7					8	\r	\n

When we return to the top of the loop, `eof()` is still false, so we read an int. Because the read is done with a >> we start by discarding whitespace (a single blank), then read and print the int (2).

The input remaining is

\r	\n		3			4	\r	\n	5				6	\r	\n		7					8	\r	\n

When we return to the top of the loop, `eof()` is still false, so we read an int. Because the read is done with a >> we start by discarding whitespace (a carriage return, a line feed, and a single blank), then read and print the int (3).

The input remaining is

		4	\r	\n	5				6	\r	\n		7					8	\r	\n

Now let's jump forward in time a bit. What will be the input remaining just after we read and print the last number (8)?

The input remaining is

\r	\n

When we return to the top of the loop, `eof()` is *still false*, because there are unread characters still left in the file. So we try to read an int. Because the read is done with a >> we start by discarding whitespace (the final carriage return and line feed), then try to get an `int` from what remains. Alas, there are no characters left, so the attempted read of an `int` fails, and the int is left unchanged.

How do we fix this? Well, one option is to simply test all our inputs to be sure they succeed. Change the program to:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream input ("data2.txt");
    while (!input.eof())
    {
        int k = -12345;
        input >> k;
        if (input) // input.good()
        {
            cout << k << endl;
        }
    }
    return 0;
}
```

Compile and execute. Does that take care of the problem?

Now in that version, the `eof()` test is pretty much a waste. We might as well have written

```
#include <iostream>
#include <fstream>

using namespace std;
```

```
int main()
{
    ifstream input ("data2.txt");
    while ( !input )
    {
        int k = -12345;
        input >> k;
        if (input) // input.good()
        {
            cout << k << endl;
        }
    }
    return 0;
}
```

which, in turn, suggests a somewhat more elegant approach might be to simply write the loop to run as long as we can successfully read more integers: Compile and run this version:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream input ("data2.txt");
    int k = -12345;
    while (input >> k)
    {
        cout << k << endl;
    }
    return 0;
}
```

Note that this exits as cleanly as the last version.

2.2 Strings

Suppose that we wanted to read some text from a file, processing it word by word. This might be enough to do the job:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    ifstream input ("data3.txt");
    int count = 0;
    string word;
    while (input >> word)
    {
        ++count;
        cout << "Word #" << count << " is \"" << word << '"' << endl;
    }
    return 0;
}
```

Compile this, then create a data file `data3.txt`:

```
He took his vorpal sword in hand:
  Long time the manxome foe he sought --
So rested he by the Tumtum tree,
  And stood awhile in thought.
```

Run the program.

- Not surprisingly, because we are reading with >>, any whitespace characters at the front of an input are ignored. (The reason I printed quotation marks around each word was so that it would be obvious if there were any blanks inside.)
- But note that when reading `strings` via >>, the input also *ends* at the first whitespace character after we have accumulated at least one character from the input. Hence, the `string` version of >> is naturally "word by word".
- But we have to stretch the definition of a "word" a bit. Note that where a word of the input is followed or even preceded by punctuation, those punctuation characters are included in the "word" that we read. If you look closely, you may even find one "word" consisting entirely of punctuation.

3 Line-By-Line Inputs

Now, let's change our string handling problem a bit. Suppose that we wanted to read a list of names.

Change the program just a little:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    ifstream input ( "data4.txt" );
    int count = 0;
    string  name ;
    while (input >>  name )
    {
        ++count;
        cout <<  "Name #"  << count << " is \"" <<  name  << '"' << endl;
    }
    return 0;
}
```

and add a new data file, `data4.txt`:

```
John Doe
Jane Smith
  J. Q. Public
  Donald Mac Gillavry
Sandra de la Fiore
```

Compile the program and execute. Note that this style of input does not work very all when we want strings (names) that have embedded blanks. And, because the number of distinct "words" that make up a name may vary considerably, we can't get by with any cheap tricks like reading a pair of words and stitching them together to form a firstName-lastName combo.

This is a place where we need some formatting clue to tell us where each input ends. In this case, that formatting cue is the end of the line. To read entire lines, we use the `getline` function. Change the program to:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    ifstream input ("data4.txt");
    int count = 0;
    string name;
    getline (input, name);
    while (input)
    {
        ++count;
        cout << "Name #" << count << " is \"" << name << '"' << endl;
        getline (input, name);
    }
    return 0;
}
```

Compile and execute.

- You should be able to see that this does indeed read an entire line each time, no matter how many blanks are inside.
- However, unlike the >> operator, the `getline` does *not* skip whitespace before starting to read.
  - If we don't like spaces at the beginning of our string values but we have data that, as in this file, is not left-justified, we would need to add some code to strip out the offending blanks after reading the line.
- Although leading blanks and tab characters are not removed from the input, the line feed and (if in Windows) carriage return at the end of the line are discarded.

Sometimes we will need to pay attention to input lines even when we aren't reading strings. For example, suppose that we wanted to read a file like this (create this as `data5.txt`):

```
1 4 7
2
100 5.7 -6.0 2 2 3 3 1 2 0.5
1 0 0 0 0 0 6
```

and to print the average of the numbers in each line. If we use our free-format approach to reading numbers with >>, we have no good way to know when we have reached the end of a line.

The way to handle situations like this to break it down:

```
while (there are more lines to read)
{
   read a line;
   count and add up the numbers in that line;
   print the sum;
}
```

which leads us to the beginning of a program like this:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    ifstream input ("data5.txt");
    string line;
    getline (input, line);
    while (input)
    {
        int count = 0;
        double sum = 0;

        // Count and sum the numbers in the line
           ⋮

        if (count > 0)
           cout << "Average over " << count << " numbers is " << sum/count << endl;
        else
            cout << "Cannot average 0 numbers." << endl;
        getline (input, line);
    }
    return 0;
}
```

So how do we count up and sum the numbers in a line (a string)? How do we even *get* the individual numbers from a string?

The solution is another kind of input stream called an `istringstream`. An "istream" is an input stream that reads from an unknown device. An "ifstream" is an istream that reads from a ffile. So an istringstream is an istream that reads from a string. Instead of giving it a file name, we give it a string value and read right out of that string.[1]

[1] And, yes, there is a related class `ostringstream` that lets us write data into a string.

Once we have an `istringstream`, we read from it just like we read from any `istream`. In this case:

```
        istringstream lineIn (line);
        double num;
        while (lineIn >> num)
        {
            ++count;
            sum += num;
        }
```

Putting it all together gives us this program:

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

using namespace std;

int main()
{
    ifstream input ("data5.txt");
    string line;
    getline (input, line);
    while (input)
    {
        int count = 0;
        double sum = 0;

        // Count and sum the numbers in the line
        istringstream lineIn (line);
        double num;
        while (lineIn >> num)
        {
            ++count;
            sum += num;
        }

        if (count > 0)
           cout << "Average over " << count << " numbers is " << sum/count << endl;
        else
            cout << "Cannot average 0 numbers." << endl;
        getline (input, line);
    }
    return 0;
}
```

Compile and execute this program.

4 Problems for You

Based on what you have learned here, consider the following questions. You might want to try coding up some examples of your own to see if your solution works. Feel free to discuss the answers in the Forum, but try to take some time thinking about them before reading what others may have said.

1. You are processing a company's inventory and the files consist of product identification codes followed by the number of parts on hand:

   ```
   001A14  105
   001B14  24
   002B27x  79
   ```

   and so on.

   Assume that an identification code never contains blanks. How would you handle this input?

2. You are processing a file that contains customers' names and the number of times the customer has visited the store this year. How would you process the data if it looks like this?

```
5 Dr. John Doe
21 Ms. Jane Smith
6  J. Q. Public
0  Donald Mac Gillavry, Capt. USAF
14 Sandra de la Fiore
```

3. Same problem, but the data is arranged like this:

```
Dr. John Doe 5
Ms. Jane Smith 21
J. Q. Public 6
Donald Mac Gillavry, Capt. USN 0
Sandra de la Fiore 14
```

   - What makes this arrangement of the data so much harder to deal with?
   - It's tempting to approach this problem by reading word by word, first trying to read something as an `int`. If that read succeeds, that's out number and we were already done with the name part. If, on the other hand, we detect that `fail()` was set, we read it as a string instead and add it to the name.
     – What happens if Donald recommends our store to a friend who is a 2nd Lieutenant?
   - Can you think of a better approach?

4. Suppose that we wanted to compute the average of several data sets that were combined like this:

```
1
4
7
End of Set
2
End of Set
100
5.7
-6.0
```

```
2
2
3
3
1
2
0.5
End of Set
1
0
0
0
0
0
6
End of Set
```

How would you process this input?

See if you can work out two different approaches, one using `istringstream` and one that does not.