COSC 2P95 – Lab 4 – Section 01 – IO

As with all other labs for this course, you'll still be using Linux for this lab. As such, reboot and log back in if necessary (refer to Lab 1 for details).

In this lab, you'll be using both files and the console for basic IO functionality.
Before getting any further, don't forget that reference documentation is absolutely vital for using libraries.
A (very) non-exhaustive list includes:
http://www.cplusplus.com/doc/tutorial/files/
http://www.cplusplus.com/reference/ios/
http://www.cplusplus.com/reference/library/manipulators/ — just for manipulators
http://en.cppreference.com/w/cpp/io/manip — because two references are better than one
http://en.cppreference.com/w/cpp/io/basic_ios
http://en.cppreference.com/w/cpp/io
http://en.cppreference.com/w/cpp/io/ios_base/openmode — useful for opening files
http://en.cppreference.com/w/cpp/io/ios_base/iostate

Note: It is assumed that you've already reviewed the examples covered during lecture. There's no time to revisit every little detail within the lab, so definitely make sure to remember that content as well when studying or working on exercises.

Reminder: you've been `using` the std namespace, but that isn't enough to cover everything. You may find yourself needing to prepend, for example, `ios::` in front of some terms. Neglecting namespaces and forgetting to `#include` the correct headers are, by far, the most common mistakes you're likely to make.

Simple console IO

Because you're running your programs from a terminal, it makes sense to be well-versed in the command-line environment. This includes our standard IO streams, as well as redirection (feel free to play with pipes as well, but you won't be compelled to use them for this lab).

Start by creating a fresh document, and include the `iostream`, `iomanip`, and `fstream` headers. You won't need all of them yet, but you will eventually, so let's ensure that we don't forget any of them.
As always, feel free to add a line for the std namespace.

Write a program with at least three procedures:

- One to print a greeting (e.g. Hello)
  - Use `cerr` (instead of `cout`) for this output
- One to print a goodbye
  - Again, use `cerr`
- A main procedure, that starts with the greeting, contains a loop, and then ends with the goodbye before exiting
  - The loop simply reads entire lines of text from the input, to then print them out
  - The loop terminates when EOF is encountered
  - Use `cout` for your output

What you might notice is that the program will use two different output streams, depending on whether it's putting out data, or notifying the user.
When you run it, assuming you compile it to, say, `squeak`, try running it as follows:

```
./squeak 1>data.txt 2>log.txt
```

In addition to what's included at the beginning, you might also find these references useful:
http://www.cplusplus.com/reference/istream/istream/get/
http://www.cplusplus.com/reference/istream/istream/read/

After the lecture examples, this should be easy, but let's still look at the components a bit:
First, the basic approach will be to have a `while` loop, that continues looping so long as the fail and eof bits have not been set. The easiest way to test this is to use cin itself as a boolean condition.
i.e. `while (eof)` will succeed, so long as the end of file hasn't been reached. Of course, you can also explicitly test for eof(), good(), etc.

When reading in, you have a few options:

- `getline(cin,s)`
  - This assumes `s` has been declared as a `string`
- `cin.getline(buffer,40)`
  - This assumes `buffer` is a character array, with length of at least 40
- `cin.get()`
  - Returns a single character
- `cin.get(c)`
  - Assuming `c` is a char, this uses a *reference* to `c`
- `cin.read(buffer,40)` (or `cin.read(&c,1)`, but please don't)

Of course, what you give to `cout` will depend on what you used for holding the input.
Also, considering some of these will fail if you try entering excessively-long lines, you might wish to not rely on a presized buffer. Also, if you're using a technique that strips away the newlines (e.g. getline), then make sure to include `endl`'s when printing.

As a final bit of fun, let's use the contents of your generated data file as the input for a second execution!
```
./squeak 1>newdata.txt 2>log.txt <data.txt
```

Note that, depending on how you were reading/writing, it may add an additional newline or so. At this level, it doesn't really matter, but that's something to keep in mind for when it does.
If you should need it, don't forget that the `gcount` function tells you how many bytes were read.

File reading

To read a file is pretty simple. You can use an `ifstream`, or simply an `fstream`.
In any case, we can use the same basic `istream` behaviours with them.
Create a file, containing the following:
```
14    8 22 lemur     100
```
(it doesn't actually have to be that exactly. Just include some integer values, separated by spaces, tabs, or newlines, and also throw in a normal word somewhere towards the middle)

Let's try to read each of those numbers in, and recover when it fails (because of the *lemur*).

Give it a try, and then flip to the next page to see a minimal solution (including explanation).

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

int main() {
     int val;
     fstream fin("filein.txt",ios::in);
     while (fin) {
          fin>>val;
          if (fin.fail()) {
               fin.clear(ios::eofbit&fin.rdstate()|ios::goodbit);
               fin.ignore(1);
          }
          else cout<<val<<endl;
     }
}
```

Let's look at this piece by piece:

- Why use an fstream instead of an ifstream?
  - Why not? (There's really no pressing reason to use one over the other)
- What happens if it encounters a space instead of a tab?
  - Whitespace is whitespace
- What happens when it encounters the lemur?
  - The failbit is set. We need to clear it
- What else?
  - We need to clear out the lemur
    - We don't just want to stop at a newline this time, since we could have any number of delimiters
    - Instead, let's just clear out the first 'l'
      - And the the 'e' on the next iteration
        - And then the 'm'…
          - etc.
- That `clear` line is designed to retain the eof flag, if encountered

Now, let's make one small incremental change:

- Take the entire contents of that main procedure, and put them into a separate procedure.
- Call that new procedure from your main.

What's the point, right? Does the exact same thing?
As one additional change:

- Move the fstream declaration line back to main, and pass it as an argument to the procedure

e.g.

```
void wokka(istream &fin) {
     int val;
     while (fin) {
          fin>>val;
          if (fin.fail()) {
               fin.clear(ios::eofbit&fin.rdstate()|ios::goodbit);
               fin.ignore(1);
          }
          else cout<<val<<endl;
     }
}
```

```
int main() {
     fstream fin("filein.txt",ios::in);
     wokka(fin);
}
```
(note that the parameter is an `istream`)
So, what's the difference? Still the exact same?
One final change:

- Change `wokka(fin)` to `wokka(cin)`

Of course, it's silly to still open the file, but that's not the point. The point is that we switched from using file into to console by changing a single character.
What this means is that the various iostreams are actually pretty flexible (and substitutable).

Formatted output

We've already covered quite a bit of this during lecture, so let's just revisit one aspect of manipulators and formatting options: column width.
Let's start with this:

```
void cleanOutput(int values[16]) {

}

int main() {
     int values[16];
     for (int i=0;i<16;i++) {
          cout<<"Next value: ";
          cin>>values[i];
     }
     cleanOutput(values);
}
```

What we wish to achieve is simply: Display four rows, of four columns each. Each column should be the width of the greatest width of any column, plus 1.
So, with values of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16, each column would be allocated 3 characters of width.
How could we inspect the values to determine the highest width? Perhaps a function? I'll leave that part to you.
Assuming you wrote a function that lets you say:

```
int width=inspectWidth(values);
```

All you need do for your formatted output is to use the `setw` manipulator:

```
cout<<setw(width)<<values[…
```

Submission Task

Open up your text editor (presumably gedit). Type in a few lines of text, and then save it as a .txt file.
For mine, I'll be assuming a filename of `sample.txt`.
If you were to open that file up in notepad (on Windows), you might not like how it renders.
We haven't talked very much about newlines.
The Unix standard is to use a single LF (linefeed) character.
The windows standard is to use a CR (carriage return), followed by an LF.
What this means is that a plaintext file written in one will sometimes not be 100% compatible with the other.

So, what's the solution?
There's a program called `unix2dos` that converts a file with LF line endings into a file that uses CR+LF. Similarly, `dos2unix` does the reverse.
You may not have them installed on your lab computer, so just log into sandcastle and try the following (from the same folder):

```
ls -l
unix2dos sample.txt
ls -l
dos2unix sample.txt
ls -l
```

You should see that the file size increases slightly from unix2dos (because it's adding one byte per newline), and it shrinks back down after dos2unix.

So, for this task: you'll be writing a very simple version of dos2unix (call yours d2u) and unix2dos (u2d).
You shouldn't have too hard of a time writing it, so long as you've been following everything above.

All you need to do is to read one character at a time, and cout the same character, with one caveat:

- When converting dos→unix, ignore any character that matches 13 (or 0x0D, if you prefer hex)
- When converting unix→dos, whenever you encounter a 10 (or 0x0A), output a 0x0D char (you may need to cast), followed by a 0x0A char

You may wish to open in binary format.
You may accept command-line parameters, or just use hardcoded filenames.

- For the sake of simplicity, I'd advise against reading from and writing to the same file

Don't forget to close your steams when you're done.

If you'd like to verify that it worked, there are two particularly easy ways. Assuming the new file is new.txt:

- `hexdump sample.txt` followed by `hexdump new.txt` will display hexadecimal representations of both files in their entireties. It's pretty easy to spot the 0a's (linefeeds) and the 0d's (carriage returns)
- `md5sum sample.txt` and `md5sum new.txt` will produce md5 hashes, which act as a de facto checksum

Submission
To demonstrate that you understand the content from this lab, simply show your text conversion tool to your lab demonstrator.