Lab 10
Weather stats

10 November 2015

The drill this week will give you some practice with using `getline` and with stringstreams. The idea is that the data file represents a log file or diary, one entry per line, and your program can then process it. Come to lab on Tuesday either with it completed or with a specific written question in your notebook identifying which drill step you got to and what about it you're stuck on.

1. Start by creating a test file that has at least five lines in the format

   ```
   11 5 Got up early, feeling good!
   11 5 Finished a bunch of work in the afternoon
   11 6 Overslept my alarm. :(
   ```

   Each line should start with a month and day, and then the rest of the line has a short message.

2. Write a program that reads each line of the input, as long as there's input, and prints how many lines there were in the log file. Note that we can put a `getline` call into a `while` condition just like we could with `>>`:

   ```
       while (getline(cin, line))
         //...
   ```

3. Modify your program to print just the dates on which a log file entry was made. To do this, you'll make an `istringstream` to process the line—similar to the code we wrote in class—and then from the line, read the two numbers (month and day) into int variables.

4. Modify your program to reformat the input into the following output format:

   ```
   11/05:  Got up early, feeling good!
   ```

   Note that the way you can get the "rest of the line" in the stream is to make a call to `getline` using your stringstream instead of `cin`.

5. Modify your program to only print the *first* log entry from a particular day. You can assume all the dates are in order already; this will thus involve remembering what date was used in the previous line, and only printing if the current line has a different date number.

6. Modify your program again so that if there are multiple log entries for a particular date, it prints the first one followed by

   ```
   (subsequent entries omitted)
   ```

   It should only print that message once per date, even if the date has more than two total entries.

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next section about vim commands). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, go on to the next section.

Vim FOTD: Miscellaneous commands

A few commands you might find handy. Remember, all commands are case sensitive and work in command mode.

`J`	Join this line and the next
`rX`	Replace this character with X, for any value of X

Commands that initiate insert mode:

`s`	Replace this character
`cc`	Replace this whole line *
`I`	Insert at beginning of line (but after whitespace)
`a`	Insert after this character
`A`	Insert at end of line
`o`	Add a line after this one, and insert there
`O`	Add a line before this one, and insert there

`==`	Reindent this line *
`<<`	Shift this line to the left *
`>>`	Shift this line to the left *
`.`	Repeat the previous command

All commands marked with a * are given in "this whole line" form, but (like `dd` and `yy`) can also be used with any movement command, so for instance `cw` replaces from here to the end of this word with whatever is typed in insert mode, `=G` reindents from here to end of file, and `>%` shifts to the right everything between this line and the matching curly bracket.

Also, these commands and every single other command in vi can be preceded with a number (can be more than one digit), which typically means "repeat N times", though the exact interpretation varies from command to command. `15G` will go to line 15 of the file, for example, and `3rQ` will replace the next three characters with the character `Q`.

Stats revisited

Back in Lab 6, the intent was to give you a first exposure to functions, and in particular, functions that processed vectors of stuff—in that case, vectors of doubles. This week, you'll revisit the basic task, this time with the "stuff" being structures containing bundles of data.

To get started on this part of the lab, copy some files into your directory. The task of the week will involve processing weather data, so first, you'll copy a set of weather data, found in

```
/home/shared/160-3/weather-big.txt
/home/shared/160-3/weather-small.txt
```

These are data about the weather for the month of August 2009 in Galesburg, IL (where I worked at the time). Each line of the file contains information about the date and time it represents, as well as the temperature (in degrees Fahrenheit) and the wind speed (in miles per hour). This is a sample line illustrating the format:

```
08 02 2009      18 00   76      12
```

The date is first, then the time: this line represents August 2, 2009, at 18:00 (6pm). The temperature was 76°F, and the wind was blowing at 12mph. The "big" file contains measurements for every hour that month; the "small" file contains two days. (You're also encouraged to create your own files, even smaller, for testing purposes.)

Also, copy the stats-related files from your Lab 6 directory into your Lab 10 directory: `stat_functions.h`, `stat_functions.cpp`, `test_stat_functions.u`, and `run_stats.cpp`. It's ok if they didn't work in Lab 6 and don't work now—you won't be graded on them again—but having them here will let you conveniently refer back to what you did there. (If you got those parts of Lab 6 working and wish to make use of them this week, you may, but if not, don't worry about it.)

Defining and reading the struct

Based on the description above, and using the examples of `Token` (from last week's lab and Ch. 6), `Card` (from lecture), and `Point` (from the Chapter 10 drill last Wednesday), define a `struct` (or `class` using `public:`) called `Weather` that is capable of holding all the data on each line of our data file. This definition should go in a file `weather.h` (which will also be where the related function headers will eventually go).

In a file called `run_weather_stats.cpp`, make sure to `#include` the file you just created, and then define a `main` that reads the name of a data file from `cin`, then opens that file and reads weather data line by line. Each line should be read from the file using `getline`, and processed using a `stringstream`; and a `Weather` value corresponding to each line should be created and put in a vector.

In order to write this, make sure to avail yourself of all the examples we've seen of programs with similar tasks and features: in previous labs, in the book, in the shared course directory, and on the board during class.

In a loop purely for testing purposes, print out just the temperature from each `Weather` value in the vector. This will help you confirm that everything is being read and stored that is supposed to be. Once you have confirmed this, comment out (don't delete) the loop.

Functions to process vectors of weather

For this part of the lab, write functions that take vectors of `Weather` and process them. In some cases the result would be a single number; in others it could be a `Weather` value or even a whole vector of `Weather` values.

- A function `average_temp` that computes the average temperature of all the `Weather` values in the given vector.

- A function `min_temp` that computes the minimum temperature of all the `Weather` values in the given vector.
- A function `max_temp` that computes the maximum temperature of all the `Weather` values in the given vector.
- A function `hottest_time` that finds and returns the `Weather` (which includes date/time info) that represents the hottest moment in the given vector. (If there are multiple equal "hottest" moments, any one of them could be returned.)
- A function `noon_data` that filters and returns a vector of all those `Weather` values in the given vector that represent a measurement taken at noon on some day.

In each case, the function header should be typed in to `weather.h` and then the function should be defined in `weather.cpp` and called in `run_weather_stats.cpp` to print their result. You should also clearly test each function you implement, which *may* be done in a file `test_weather_functions.u` (as in Lab 6 with the stat functions, and last week in lecture with the token/whitespace work), or it *may* be done with input and expected output files: either way is fine as long as I can tell you've tested them and can see (from your readme) how to run those tests.

Don't forget that to compile and link your program, you'll need to account for both `main` and the other functions:

```
  compile run_weather_stats.cpp weather.cpp
```

and similarly

```
  compile test_weather_functions.u weather.cpp
```

if you use a `.u` file.

Handing in

Hand in as `lab10`; it is due Monday at 4pm as usual.

Rubric (tentative)

RUBRIC

General

1	Attendance at lab with drill done or question written down
1	Documentation clear and correct on how to compile, run, test

Drill (log file)

1	Reads input line-by-line until done, prints (some part of) each line
½	Extracts dates from each line
½	Formats date in output
½	Reads and prints “rest of line” correctly
½	Identifies first line per day, excludes others
½	Prints “omitted” message correctly
½	Code has clearly been tested

Weather

½	`Weather` defined
½	Uses stringstream to process line and produce a `Weather` object
½	Reads weather data from file into `vector` of `Weather`
1	Computes min, max, average temperature using functions and prints
½	Computes hottest time using function and prints
½	Computes list of noon-time data using function and prints
½	Code has clearly been tested