Lab 6
Vectors

6 October 2015

The drill for this lab is the **Chapter 4 drill**. 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.

Notes and adjustments:

- In step 1 of the drill, it says to exit when a terminating `'|'` is entered. Instead, just keep reading as long as there's input.
- Remember that Ctrl-D terminates input if you're typing input by hand at the keyboard. If you're redirecting from a `.in` file, the end of the file terminates input without you having to do anything special.
- After step 5, **make a copy** of your `.cpp` file before you continue working; your (eventual) handin should include one program for steps 1–5, and another for the rest of the steps. Both programs should have appropriate test cases!
- Remember that `cp` is the command for copying files on the command line in Linux. Consult your Lab 0 handout if you don't remember how to use it.
- Skip steps 7 and 8 regarding units and unit conversion. (Assume that all the numbers entered are lengths in metres.)

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 sections about compiler options). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, continue on to the next section.

Command line FOTD: compile options

Actually, just two quick things: a fact about `compile`, and two things you can do with `compile` that will make our lives easier.

First, `compile` is happy to accept any number of input files at once:

```
compile sourcefile1.cpp otherstuff.cpp stillmore.cpp
```

Functions in any of those files can then refer to functions defined in the other files. There are just two ground rules:

1. Across all the files, there needs to be exactly one `main` (or a `main` replacement, as we'll see later).
2. Functions used from other files need to have what the book calls a "function declaration". (We'll see those in a little while too.)

Related to this fact, one convenient thing that we can do is tell the `compile` command to *only* compile, and not also link—which means we can check for compiler errors in a single `.cpp` file even if it doesn't have a `main` function in it, or in a single `.u` file, even if it refers to functions that aren't defined yet. We do this with the `-c` option:

```
compile -c partialprogram.cpp
```

This will not generate an executable even on success, but will give you any compiler errors lurking in that file.

The second convenient thing is that even when linking an executable, `compile` doesn't always have to generate a file named `a.out`. You can specify the name of the executable by preceding it with `-o`:

```
compile sourcefile1.cpp otherstuff.cpp stillmore.cpp -o programName
```

(Note, however, that this will overwrite whatever is in `programName`—don't accidentally type `-o sourcefile1.cpp` and overwrite your own code!) You could then call the program as

```
./programName
```

We'll see examples of that, too.

This is one of the big reasons I've been making you include "to compile" and "to run" instructions in your readme—the compile instructions may include multiple files or additional options, and depending on the compiler

options, the command to run may vary. When you use the `-o` option, let me strongly encourage you to copy the compile line and paste it into the window, or (once you've done that once) use the up-arrow to re-execute the previous command; this reduces opportunities to accidentally overwrite something.

Right now, edit your readme to add the output options "`-o ch4drill1`" and "`-o ch4drill2`" in the "to compile" lines of your drill work, and then change the "run" instructions to

```
./ch4drill1
```

and

```
./ch4drill2
```

(with similar changes to any testing instructions).

Vector functions

For the rest of the lab, you'll write code using functions that process vectors. While the code will be similar in several ways to what you wrote in the drill, leave the drill as its own separate source files generating their own separate executables.

The `sum` function

The first function you'll type in is `sum`, which is pretty similar to what I wrote in class yesterday, but combining the function stuff and the vector stuff into a single task.

First, the function declaration, which gives only the header of the function. It will go in a separate "header" file, which ends in `.h` to signify its role. Call it `stat_functions.h`, and give it the following contents (just one line for now; you will add more later).

`stat_functions.h`

```
double sum (vector<double> nums);
```

It says that the `sum` function, whenever it gets defined, will take one parameter that is a `vector` of `double`, and it will produce a `double` as its return value.

Next, a test file to specify what it's supposed to do. It follows the same format as the test file I did in class yesterday.

test_stat_functions.u

```
#include "std_lib_facilities.h"
#include "stat_functions.h"

test suite stats
{
tests:
  test sum
  {
    vector<double> one_num = { 3.25 };
    vector<double> three_nums = { 2.0, 1.5, 4.0 };

    check (sum(one_num)) expect == 3.25;
    check (sum(three_nums)) expect == 7.5;
  }
}
```

We need to write the function itself, which we'll put in a third file:

stat_functions.cpp

```
#include "std_lib_facilities.h"
#include "stat_functions.h"

double sum (vector<double> nums)
{
  double result = 0.0;
  for (double val :  nums)
  {
    result += val;
  }
  return result;
}
```

AND FINALLY, one file to actually use the functions with arbitrary user input:

run_stats.cpp

```
#include "std_lib_facilities.h"
#include "stat_functions.h"

int main()
{
  vector<double> nums;
  double input;
  while (cin >> input)
  {
    nums.push_back(input);
  }

  cout << "Sum:  " << sum(nums) << endl;
  return 0;
}
```

To compile and run this program, you'd do

```
compile stat_functions.cpp run_stats.cpp -o run_stats
./run_stats
```

and then type in numbers, pressing Ctrl-D on a line by itself to end input. You may (but don't have to) create a pair of `.in` and `.expect` files to test this, following the pattern that we've used for a while now (and that you presumably used in the drill). The reason you don't have to do that is that the function can be tested using the Unci testing system—that `.u` file I had you type in. To compile and run the test, you'd type

```
compile stat_functions.cpp test_stat_functions.u -o test_stat_functions
./test_stat_functions
```

If you typed in everything correctly, that should tell you

```
.

OK (1 tests)
```

But if it doesn't compile or if there's something wrong with the function, it'll report errors.

Right now:

- Make sure everything is typed in correctly and compiles and runs according to the above instructions
- Don't forget to add those instructions to your readme!
- In some of the newer stuff that you haven't seen before, try changing different things and re-compiling and running to see what kinds of error messages they elicit. This should remind you of Lab 1—it's exactly the same idea. You can even write down some of these new error messages to add to your list.

More vector functions

Following the general pattern I showed you for `sum`, now write `min` and `max`, which compute (respectively) the minimum and maximum value in the given vector.

- Add their headers to `stat_functions.h`. Each will be one additional line.
- Add tests to `test_stat_functions.u`. Each test will be a block basically like the "`test sum`" block, but with different expectations on what will be returned from the function call. They will go after the existing test block but before the final curly bracket (because they're still part of the test suite).
- Add lines using them and printing results at the bottom of the `run_stats.cpp` file.
- Write the functions themselves inside `stat_functions.cpp`. Their structure will be similar to `sum`, but with an `if` inside the for loop to decide whether to update the result (as in item 6 of the drill).

Something to think about for `min` and `max`: what should they do if the given vector is empty? Another issue (for `max` at least): what if the whole list is negative? Try to come up with an answer, and we'll talk about this in class (possibly after break).

Testing inexact numbers

As I mentioned in class, one difficulty in writing test cases for anything involving floating point numbers is that they can represent a computation with high precision and yet still be inexact. For instance, if we compute

```
1.0/3.0
```

the "ideal" mathematical result would be $\frac{1}{3}$, or $0.3\overline{3}$, but we can't really write either of those. To address this issue, the Unci test system (the stuff in the `.u` files) has a way to specify a floating point answer approximately, along with the error tolerance we're willing to accept; so we can write

```
check (1.0/3.0) expect about 0.3333 +- 0.0001;
```

Under the hood, this is doing the same sort of computation you did for the "almost equal" part of the drill.

Strictly speaking, almost any computation involving floating point numbers should be tested using error tolerances like this (including the ones we've already written)—feel free to go back and change the test cases I've already given you to use the plus-or-minus notation (but definitely use it in the ones involving division).

Still more vector functions

For the final version of all of this, I want you to have a simple statistics package that can identify seven standard statistics about a group of numbers:

- Sum
- Smallest value
- Largest value
- Mean (average)
- Median (see below)
- Variance (see below)
- Standard deviation (square root of the variance)

You've already got three of those. Each of these should be its own separate function—and note that a few of the ones you haven't written can *call* the ones that you have. (For instance, do you see where might a call to `sum(nums)` come in handy?)

One of the things you need to compute the median is a list sorted into numerical order. Remember that C++ provides a function to do that work for you! See §4.6.3 for some hints on computing the median (but note that the code there doesn't compute the correct median for even-length lists).

The variance of a list of numbers is defined as the average of the squares of the differences between each number and the mean of the whole list. That is, the (average of the (squares of the (differences between (each number) and the (mean of the whole list)))). So, to compute it, you'll first compute the mean, then go through the list computing differences, and squaring them, and so on.[1]

(If you're looking to generate test cases and don't have a graphing calculator handy, Wolfram Alpha is nice for this: type in a list of numbers separated by commas and it will give back a number of relevant statistics, including most of what you need here.)

Handing in

As usual, use the `handin` program. Designate this as `lab6`. Hand it in by 4pm on Monday, 19 March.

[1] For reasons you'd learn in a statistics class, variance is defined as an average (as I've done above) but on this kind of data it is computed with a denominator of $n - 1$ rather than n. For purposes of this lab, I don't care which of those two denominators you use.

Rubric

This lab's a little bit longer than earlier ones, and you have longer to finish it, so it's out of 15 points.

RUBRIC

General

1 Attendance at lab with drill done or question written down

1 Readme includes correct instructions to compile, run, test

Drill part 1

1 Loop reads one number each time, stops at end

1 Smaller/larger

1 "almost equal"

Drill part 2

½ Loop reads two numbers each time, stops at end

½ Tracks, prints smallest/largest so far

1 Stores numbers in vector of `double`

½ Prints vector of numbers

½ ...in sorted order

Stats functions

1 `sum` is correctly typed in, compiles, runs

3 `min`, `max`

- **½** headers in `.h`
- **½** test cases defined in `.u`
- **½** test case coverage
- **½** at least one defined, with loop containing comparison
- **½** both defined, with roughly correct algorithm
- **½** both work for all legal input

3 other vector functions

- **½** mean function defined
- **½** ... and well tested, including using `about +-`
- **½** median function test coverage
- **½** median function works correctly on odd-length vectors
- **½** median function works correctly on even-length vectors
- **½** variance and standard deviation functions

Don't forget to claim your rubric check!