

Lab 11

Weather stats

12 November 2024

This lab will give you continued practice working with structs and with vectors and with vectors of structs (and writing functions for them), and additionally will involve reading in data rather than solely hand-building the vectors and structs in a test suite.

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

into your own working directory for this lab.

2. Look at (but don't change) the contents of the files, then continue reading for their description.

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:

```
8 2 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, but make sure they follow the same format.)

3. Based on the description above, and using the examples from class (and the struct you wrote for last week's lab), define a **struct** 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).

4. Create a file `test_Weather.u`, make a test suite with only (for now) a `fixture` section (see below if you don't know or don't remember where that goes), and in that section make at least two examples of `Weather` objects, and one `vector` that contains those two `Weather` objects.
5. Compile it! With just the `.h` and the fixture in the `.u` there is enough to compile. No tests to run yet, but you can verify if your syntax is correct.
6. Add a function header to `Weather.h` for a function named `everBelow55` that will determine whether the temperature in a given vector of weather values ever dipped below 55 degrees.
7. Create a file `Weather.cpp` with a stub for the function you designed in the previous step.
8. Go back to the file `test_Weather.u` and add a `tests` section to your suite, with tests for the `everBelow55` function. Use the vector that you already put in the fixture, and add another vector so that you can more thoroughly test the function.
9. Compile again—note that your compile line will need to include both the `.u` file and the newly made `Weather.cpp`, or you'll get a linker error. (Don't forget to update the readme!) Remember that it's ok that the tests are failing now (and if they're not, they're broken tests, and you should resolve that).
10. Finish writing the function.

I'll be circulating around the lab to answer questions. If you're stuck on some part of the preview, ask me about that. If you're not stuck but haven't finished the preview, work on that now. If you're done with the preview, go on to the next section.

Doing more than just testing functions

For weeks now, we've been writing functions that were only barely related to each other, and writing tests for each function, and calling it a day—but is that really writing a program? It's an important skill that will be part of a complete programming experience, but let's tie it back to `main`.

To build a runnable program that is something other than just a bunch of test cases, you need to write a `main` function, as you've done before. But: it needs to be in a different file than all the functions you're testing, because—crucially—you can only have one `main` in a program, a test suite autogenerates a `main` function, and if you compile a test suite together with a file containing an explicit `main`, the compiler (specifically, the linker) will complain about it.

Right now, your directory should have a `readme` and exactly three other program files: `Weather.h`, `Weather.cpp`, and `test_Weather.u`. (There might also be an executable and possibly a `.o` file or two.) Create an *additional* file named `run_weather_stats.cpp` whose job will be only to contain a `main` function, and in that file:

1. Set up the file, making sure to `#include` the `.h` file you just created.
2. Write a `main` function that prints a startup message. Compile it to check your syntax.
3. In that `main`, read in all the pieces from one line and make a `Weather` value that bundles them up. (There are seven pieces on the line, all of them integers.)
4. Verify that it compiles. Add to the `readme` a separate set of compiling and running instructions: this one will compile with

```
compile run_weather_stats.cpp Weather.cpp -o run_weather_stats
```

and you'll be able to run it either as just

```
./run_weather_stats
```

and type stuff in manually (but why would you do that?) or as something like

```
./run_weather_stats < weather-big.txt
```

to use the contents of that file as the input to the program.

5. Back in `main`, wrap the input statement in a loop, as we saw how to do in Chapter 8, that keeps reading until we run out of input, and builds a vector of `Weather` objects.

6. After the loop ends, and you have a vector of `Weather` objects, call your `everBelow55` function and print the result.

(What should that function produce for each of my provided weather files?)

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 (mean) 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_wind` that computes the maximum wind speed of all the `Weather` values in the given vector.
- A function `coolest_time` that finds and returns the `Weather` (which includes date/time info) that represents the coolest moment in the given vector. (If there are multiple equal “coolest” 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 it should be effectively tested in `test_Weather.u`, all as we’ve done in previous labs. In addition, as each function is defined, you should add a line at the bottom of `main` in `run_weather_stats.cpp` to print its result.

Don’t forget that we’ve written functions that are very similar to these before! Feel free to refer back to earlier labs and classwork.

Handing in

Hand in as lab11; it is due Monday at 4pm as usual.

RUBRIC

General

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

Weather setup

- 1 Valid and correct `struct` definition, files compile and run
- 1 Function `everBelow55` tested and defined correctly
- 1 Reads weather data from `cin` into `vector` of `Weather`

Other Weather functions

- 1 Headers and test cases for min, average temperature and max wind functions
- 1 Definitions for min, average temperature and max wind functions
- 1 Computes coolest time using function (+ tests)
- 1 Computes list of noon-time data using function (+ tests)
- $\frac{1}{2}$ Prints output of running defined functions on input
- $\frac{1}{2}$ Tests pass OR there is a note in readme

Boilerplate that makes use of fixtures (or, just put all the examples directly into test blocks):

```
//any necessary #include

test suite some_name
{
    fixture:
        type var = value; // any number of examples to be used in test cases

    tests:
        test any_name
        {
            // any needed c++ code
            check ( value-or-expression ) expect == value; // OR...
            check ( value-or-expression ) expect true; // or false
        }

        //add additional test blocks here, as many as necessary/reasonable
}
}
```

AI policy and frequent submission

(no substantive change from the Lab 3 version of this policy)

Some use of generative AI is fine, but you a) should not paste this assignment or type it verbatim into the AI prompt, and b) should not be asking the AI for the whole program all at once. (Just like you can ask for help from a human, but should not have them write the whole program for you!) If you get help from an AI chat OR from a person, you should note that in a program comment near whatever you got from them.

Relatedly, I expect that you'll run the `handin` program relatively often, and I *require* that you do so at least 2–3 times over the course of working on the lab. As a rule of thumb, hand it in after completing each 1–2 points on the rubric. Submissions that jump straight to a final, (near-)correct version with no intervening submissions along the way *will receive little or no credit* for that part.

This is a new policy that I'm experimenting with; let me know if you have any feedback.

This document was written and prepared without the use of generative AI.