

Lab 8

Writing functions

22 October 2024

In this lab, I'll again be walking you through process (similar to Lab 3) for program design, but this time focusing on individual functions rather than entire programs. Once you've established that you can do all the steps I'll let you take some shortcuts, but for now: Follow the process!

Consider the problem of counting the number of abbreviations—i.e. “words” that end in a period—in a given vector.

1. In a file named `morefunctions.h`, write a comment that summarises what the function will do (you can reuse what I wrote or paraphrase it to make it clearer);
2. then follow that comment with a function declaration appropriate to that description. Remember that function declarations are the header line of a function followed with a semicolon, and include the return type, the name of the function, and parameters (in that order).
3. In a file named `morefunctions.cpp`, include the corresponding `.h` and also create a *stub* for that function (same header as in the `.h` file, and return any valid literal, like `0` or `-1` or `7`).
4. At this point, you should be able to compile, but not yet run, the test cases by typing

```
compile -c morefunctions.cpp
```

If that does not succeed, edit the two files until the compiler is happy.

5. Write out at least two test cases *by hand* (in a notebook is fine);
6. then, in a file named `test_morefunctions.u`, enter the boilerplate for a `.u` file (you can refer to last week's file or see below in this handout), and add a test block corresponding to the test cases you wrote by hand.
7. At this point, you should be able to compile, but not yet run, the test cases by typing

```
compile -c test_morefunctions.u
```

If that does not succeed, edit the test case file (and/or the header file, but probably the test case file) until the compiler is happy.

8. *Now* you can fully compile and run the tests:

```
compile morefunctions.cpp test_morefunctions.u -o test_morefunctions
```

then

```
./test_morefunctions
```

The compiling should succeed but the tests should fail (because you haven't properly written the function yet!). (This is a good time to hand in if you haven't yet.)

9. Now would also be a good time to *add those two lines to your readme* so you don't misremember or mistype them later.
10. Write out a brief English or pseudocode description of how the function might accomplish its task, *by hand* (in a notebook is fine);
11. then, encode this algorithm into C++ in your function definition. At any reasonable time, try recompiling and running your tests: with the compiler and the automatic tester in place, it's very easy to do so and can sometimes give you valuable feedback.
12. Don't consider yourself done until you've tested!

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

Function design

This, then, is a version of the meta-algorithm for designing programs (from Lab 3), adapted for use with developing functions:

1. Understand the problem: parameter(s)? return? description?
2. Write function declaration (doc and header, in `.h`) and stub (in `.cpp`)
3. Set up the boilerplate and write test cases (in `.u`)
4. Explain algorithm (pseudocode) including nameable values
5. Encode algorithm in C++
6. Test

When you need to write a function, following this process will help you know what to try next if you're not sure how to proceed. It's also a way to get some quality thinking done about the problem before you have to properly encode an algorithm to solve it.

Rest of the lab

For the rest of this lab, design and write the following functions. Do follow the process laid out above.

A function that counts the number of words in a given vector that are *not* abbreviations. (Note that you don't have to write this one as if from scratch—make use of the function you wrote earlier! This one doesn't need a loop if it calls the other one.)

A function that finds and returns the first even number in a given vector of integers, or -1 if the vector has no even numbers.

A function that determines whether there are any perfect squares (i.e. the square of another integer) in a given vector.

AI policy and frequent submission

(still no 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 click the Submit button/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. ← scoring note!

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

Handing in and rubric

Hand in as `lab8`. Due 4pm next Monday.

RUBRIC

1 Attendance at lab with preview done or question written down

Part 1 (count abbreviations)

- 1 Comment, header
- 1 At least a stub, compiles
- 1 Test cases
- 1 Correct definition

Rest of lab

- 1 Comment and header for one
- 1 Comment and headers for all
- 3 TCs and definitions ($\frac{1}{2}$ each)
 - 1 count non-abbreviations
 - 1 first even number
 - 1 any perfect squares

Notice that: some points are available for visibly following the design process; and some points are available for good test cases *even or especially* if those test cases are not passing.

Unit testing boilerplate

For reference, boilerplate for `.u` files:

```
#include "name of corresponding .h file"
using namespace std;
```

```
test suite some_appropriate_identifier
{
  test another_identifier
  {
    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
}
```

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