Lab 8
Writing functions

17 October 2019

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!). Now would also be a good time to add those two lines to your readme so you don't misremember or mistype them later.

9. Write out a brief English or pseudocode description of how the function might accomplish its task, *by hand* (in a notebook is fine);

10. 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.

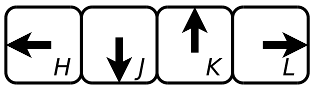
11. Don't consider yourself done until you've tested!

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next section about vim movement). 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.

Vim FOTD: movement keys

Vim responds to the arrow keys and keys like PageUp and PageDown, but there are a number of additional keys that can be pressed in command mode to move around the file. Open one of the files you have lying around and try some of them out.

Key(s)	**Movement**
`h`	Left one character
`j`	Down one line
`k`	Up one line
`l`	Right one character
`^`	To beginning of current line
`$`	To end of current line
`Ctrl-F`	Forward one page (screen)
`Ctrl-B`	Back one page (screen)
`G`	To last line of file
`#G`	To line # (e.g. 1G to go to top of file or 23G to go to line 23)
`w`	To beginning of next punctuation-delimited “word”
`W`	To beginning of next whitespace-delimited “word”
`e`	To end of this punctuation-delimited “word”
`E`	To end of this whitespace-delimited “word”
`b`	To beginning of this punctuation-delimited “word”
`B`	To beginning of this whitespace-delimited “word”
`}`	To next (batch of) blank line(s)
`{`	To previous (batch of) blank line(s)
`%`	To matching paren/bracket
`[(`	To previous unmatched left paren
`[m`	To start of current function



Some of these are more mnemonic than others, of course. The first four are not mnemonic at all, but super-convenient once you’ve got them in muscle memory, because they’re right in the home row, so your fingers don’t have to go anywhere to type them.

So what, right? Well, all of the delete commands that you learned in earlier labs were special cases of a rule: `d` plus a movement command deletes from “here” to wherever that movement goes. So, `d1G` deletes to the top of the file. And `d%` deletes everything between this paren and the matching one. Since the `p` command only pastes the most recently-deleted thing, it’s very helpful to be able to delete everything you want to “cut” all at once. Same goes for the `y` (“yank”, i.e. copy) commands. The re-indent command (`=`) is another one that works with an arbitrary movement: `=%` reindents everything between “here” and the matching paren or bracket, while `=G` reindents everything from “here” to the end of the file, and so on.

There’s no need to memorise all the movement commands right now, of course. A couple might stick, but for the rest, even if you don't remember the command, you'll remember it exists, and you can always come back and refer to this sheet.

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!)

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 negative numbers in a given vector. (NOTE: your final version of this will make use of something we'll see in class tomorrow, though you should be able to do some design work and devise a draft solution before then, if you get this far.)

Handing in and rubric

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

RUBRIC

- **1** Attendance at lab with drill done or question written down
- **Drill (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 ($^1/_2$ each)
  - **1** count non-abbreviations
  - **1** first even number
  - **1** any negative numbers

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
{

  //add individual test blocks here

}
```