Lab 7
Unit testing and debugging

10 October 2019

Below are the instructions for the drill. Pull out your hand traces, and in a few minutes we'll go over what you did with them; for now skip ahead and read the FOTD about compiler options.

Your "drill" task this week involves reading a function I have written and seeded with bugs, and doing some preliminary analysis on it.

1. First, copy this drill version into your working directory for this lab. In your working directory, type

   ```
   cp /home/shared/160/countDivisors-drill.cpp .
   ```

   (don't forget the lonely dot at the end to put it in the current directory).

2. It has several errors! Read through the file, but don't fix them yet.

3. You'll be tracing the code using a value of 12 for `numToCheck`. Before you do so, write down what you think the correct return value should be (and why).

4. By hand, trace the code using a value of 12 for `numToCheck`. Anytime you encounter a bug:

   - If it is an error that would generate a compiler error with a clear fix (e.g. misspelled identifier), "fix" the eror in your head and keep going with the trace.
   - If it is an error that would generate a compiler error with multiple syntactically valid fixes (e.g. unattached else), pick one such fix, make a note of what you chose, and keep going with the trace.
   - If it is an error that would crash the program at that point, stop the trace and read the next drill step before continuing.
   - If it is an error that you happen to spot, that you know would generate a wrong final answer but would let the program continue from that point, *don't* fix it yet, treat it as-is, and keep going with the trace.

5. Anytime you get to a point in your trace where either the function finishes or the program would crash: make a note of it; make a guess at how to fix it (and write down the guess); and then start a *fresh* trace with that fix in place. Don't just restart the trace in the middle with the changed lines!

6. If you get through two or three traces and still have not fixed everything, that's sufficient for now; there will be more on this to come later in the lab.

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 (we'll see those in a minute), 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 why 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.

As you go through this lab, take note of how I've written the instructions in the provided readme—you can use them as a model for your own versions in future labs.

Setup for the rest of the lab

In your working directory for this lab, type the following to copy several files you'll need for the lab:

```
cp /home/shared/160/lab7/* .
```

(again, don't forget the lonely dot at the end to put it in the current directory).

There are four files here. Consider them in this order:

1. The readme. Read it.
2. The `.h` file. This is a place to collect information on what functions are *going to be* defined. Read it; one of the functions is the one from the drill, but you'll be working with all three.
3. The `.cpp` file. This is where the functions are defined. At the top is the (broken, buggy) function you saw in the drill; there is a stub

for the second function and a (buggy) implementation I wrote for the third. For now, just quickly skim this file (you'll come back to it later).

4. The `.u` file. This is a kind of file you've not seen before. Open it in vim and then continue to the next section.

Unci files and unit testing

Unit testing is the idea that we can (and should!) write test cases not just for the entire program overall, but also for individual pieces (units) of it. I've written a framework called Unci that supports unit testing with a clean interface; this is not universal-standard C++ but is similar in flavour to what happens in other unit-testing systems.[1]

In this lab, you won't be building a `.u` file from scratch, just reading (and later, adding to) the one I've provided, so rather than rigorously define the Unci syntax here, I want you to read the file (and its comments) to get a sense of its layout; and as you get the different functions working, and run the provided compile command and run the test cases, refer back to the `.u` file and ask lots of questions. (We'll also be talking about this in class, of course).

Once you've spent some time reading through the `.u` file—don't worry about deep understanding yet—continue on to the next section.

Fixing countDivisors

As you saw in the drill, countDivisors has a few bugs. Fix them—and each time you make a change, leave a comment in the code OR write a line in the readme to describe the error and how you fixed it. After you think you've fixed each one, recompile the tests and run them.

Writing countWords

I've provided a declaration for `countWords`, and a stub definition, and a bunch of test cases to help define its behaviour. Write the function. Focus

[1] Many unit testing frameworks exist, but the ones for C++ are often clunky, awkward, and highly unforgiving to the beginner. Unci is better. :)

first on getting a version that works with the "basic" test cases; move on to the "zero" and "challenge" parts only once the "basic" block passes.

Testing and fixing secondHighest

The third function compiles and runs and seems to work, but the provided test cases are inadequate and the function is subtly buggy. Add test cases that more thoroughly test the function; don't change the function until you've added a test case that the existing definition *fails* and that thus demonstrates the bug(s). Once you have such a test case, *then* work on fixing the function.

Handing in and rubric

Hand in as `lab7`.

Rubric

RUBRIC (Tentative)

1 Attendance at lab with drill work visibly in notebook
countDivisors
1 Compiler errors have been resolved
1 Results are correct
1 All changes are marked
countWords
1 Loop to access all characters in phrase
1 Handles "basic" cases correctly
1 Handles "no spaces" cases correctly
1 Handles "challenge" cases correctly
secondHighest
1 Test case that causes original secondHighest to fail
1 Fixed secondHighest to work in all cases that meet preconditions