1 Lab 10

Date: Apr 16, 2020

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a # character to end-of-line.

1.1 Aims

The aim of this lab is to introduce you to testing. The lab will give you some exposure to the following topics:

- Unit testing.
- Test driven development (TDD).

1.2 Background

Testing software has always been done by conscientious developers but it was only around the start of this millenium that **automated** testing became required for professional software development. This takeoff may be due to the popularity of the junit framework as well as the popularity of agile techniques for software development.

All developers test when writing code, but those tests are often manual. It is very time consuming to set up some kind of driver program for the tests (especially in a language like C++ without a natural REPL *read-eval-print loop*. When writing the code, the developer has a thorough understanding of the code and knows what behaviors should be tested. That may not be the case when another developer needs to change the code, or even when the original developer needs to update the code a few months later.

The problem is that the knowledge about the details of the code was transient in the memory of the original developer. Capturing that knowledge as documentation is usually impractical unless there is a professional documentation team. In any case, the problem with documentation is that it often gets out-of-date and having incorrect documentation is worse than having no documentation. Automated tests provide a happy medium in that they serve as a **living document** capturing the knowledge of the initial developer.

Refactoring is the process of changing code to (hopefully) improve its quality while preserving its behavior. For example, *extract function* which simplifies a complex function into several simpler functions is one of the many refactorings available in any programmer's catalog of refactorings. One of the major advantages of automated testing is that it makes it possible to refactor code without fear of breaking it.

We will be using a C++ testing framework called catch2. It has the advantage that it does not require any installation as the entire package is in *this header file*. The disadvantage is that the header file is rather large and consequently compilation is noticeably slower.

1.3 Exercises

1.3.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab10` branch and a new `submit/lab10` directory. Start a `script` session to log your interaction into a `lab10.LOG` file.

You will be doing all your work in your `submit/lab10` directory:

```
$ cd ~/i240?/submit/lab10
```

Copy over the exercises directory:

```
$ cp -r ~/cs240/labs/lab10/exercises .
```

The rest of this lab assumes that you are using `bash` as your shell.

1.4 Exercise 1

This exercise just introduces you to the idea of testing using an example taken from the the *Catch2 Tutoral*. The 1-fact directory contains a trivial factorial program with the usual recursive implementation in fact.cc with its specfication in fact.hh. A trivial main program allows running the program using command-line arguments:

Compile and run the program. Use the Makefile in the parent directory:

```
$ make -f ../Makefile main
$ ./main 2 3 5 10
```

There is a catch2 test program in test-main.cc:. Open it up in an editor; you should find it quite readable. Compile and run it (as warned above, the compilation is quite slow):

```
$ make -f ../Makefile test-main
$ ./test-main
```

We have written our factorial in a modular way with a separate specification and implementation file. This module is referred to as a unit and our tests constitute **unit tests**.

Unfortunately, there is a bug even in this trivial factorial program. Run `./main 0`. Do you notice that the value produced is wrong?

1. Add a test to test-main.cc to capture a test for this bug. Recompile and rerun the tests. You should now get a test failure.
2. Fix the bug in fact.cc. Recompile and rerun the tests. All tests should succeed.

1.5 Exercise 2

This exercise introduces you to TDD **Test Driven Development**. The idea is that you first write tests before you write code, so that the tests will fail; then you write code to make the failing tests green and iterate until you have implemented all the requirements.

In this exercise we will illustrate TDD by a simple example to evaluate the sum of single decimal digits. Specifically, our requirements are:

- Given a string containing digits separated by `+` signs, return an integer representing the sum of the digits.
- Any whitespace within the string should be ignored.
- If at any point a syntax error like consecutive digits or `+` characters is encountered, then the evaluation should terminate, returning the value accumulated so far.
- If at any point a character other than a digit, whitespace or `+` is encountered, then the evaluation should terminate, returning the value accumulated so far.
- Accumulation of the digits should start with an initial value of 0.

Change over to the 2-tdd directory and compile the code using the `Makefile` in the parent directory.

1. The compilation should not generate any error messages, but will generate a warning message about a missing return value in `digitSum()`.

   Compilers are integral parts of test toolsets and you should **never** ignore compiler warnings. To fix this particular problem, we need to return a value. Looking at the requirements, we should initialize a accumulator to 0 and return its value. So add the following code to `digitSum()`.

```
int acc = 0;
return acc;
```

You should now get a clean compile.

2. Now write a test for this situation. Specifically, insert the assertion `REQUIRE(digitSum("") == 0 );` into the `test-main.cc` file. Compile and run tests using `./test-main`. All tests should now pass.

3. Add the assertion `REQUIRE(digitSum(" ") == 0 );` into `test-main.cc` file and compile. When you test, both tests should pass.

   Note that we have not written any code yet to handle nonempty strings. So did we just get lucky?

4. Add the assertion `REQUIRE(digitSum("7") == 7);` into the `test-main¬.cc` file and compile. When you test, this test should fail.

5. Fix the failing test. Insert code in `digitSum()` to check whether the first `char` in `str` is a digit (use `isdigit()` after including the >cctype> header. If it is, convert into a digit by subtracting `'0'`‘ from it (this was covered earlier in the course). Since we are going to be looking at the rest of the characters in `str` using a loop, we might as well start out with a loop:

   ```
   for (const char* p = str; p; ++p) {
    int c = *p;
    if (isdigit(c)) acc += *p - '0';
   }
   ```

   Compile and test. It fails miserably. Turns out there is a bug in the above loop. Fix and retest.

6. Now write a test for `digitSum(" 5 ")`. Does it pass? Why?

7. Now write a test for `"7 + 2"`. It should fail as we have not handled `+`. Make the `+` work by simply ignoring it. Compile and test. It should pass.

8. Add a test for `" 7 + 2 + 4"`; it too should pass. Are we done?

9. Unfortunately, we still need to handle the negative requirements of checking for erroneous characters and returning the accumulator returned so far. Add in a test to check for this situation: `digitSum(" 8 + x + 4") == 8`. It should fail.

   So add in a test to break out of the loop if the current character is not a digit, whitespace or `+`. Test to verify that all tests pass.

10. We still have a requirement that we should break out when we have a syntax error like consecutive `+` characters or consecutive digits. So add tests for these situations. You should find that they fail.

11. Fix the problem by adding a flag: `wantsDigit` which is initialized to **true**. Pseudo-code for the body of the loop, assuming that `c` contains the current character:

    ```
    if (isdigit(c)) {
      if (wantDigit) {
        acc += c - '0';
        wantDigit = false;
      }
      else { //expecting +
        break;
      }
    }
    else if (c == '+') {
      if (wantDigit) break;
      wantDigit = true;
    }
    else if (isspace(c)) {
      continue;
    }
    else {
      break;
    }
    ```

    Verify that all tests pass.

Note that this code using a flag is pretty messy. Now that we have tests, we could be more confident about attempting a refactor to clean it up.

1.5.1 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab10.LOG` in your `lab10` directory. Add all your files to git and commit. Then merge your `lab10` branch into the `master` branch and commit your changes.