CS 1XC3 Lab 7 : Testing and Makefiles

Mar. 13th-17th, 2023

Contents

Introduction

Over the course of this lab activity, you will create a makefile to compile code and conduct automatic testing. You will also write C programs which use multiple files and command line arguments.

Setup

- To complete this activity, you will need to make a mirrored clone of the following github repository:
  - `https://github.com/Zed-devp/Lab7.git`
- Simply forking this repository will not allow you to make it private. Please follow the following steps from whatever bash prompt you have set up:
- Create a bare clone of the repository.

```
$ git clone --bare git@github.com:Zed-devp/Lab7.git
```

- Create a new private repository on Github, and call it "1XC3_L07".
- Mirror-push your bare clone to your new private repo.

```
$ git push --mirror git@github.com:<your username>/CS1XC3_L07.git
```

- Remove the temporary local repository you set up in step 1.
- You can now clone your 1XC3_L07 repository on your local machine.

```
$ git clone <URL of your private 1XC3_L07 repo>
```

- Next, add 1XC3_L07 as a submodule to your CS1XC3 master submission repository in a new "L07" directory.

```
$ git submodule add <URL of your private 1XC3_L07 repo>
```

The repository contains two directories, "src" and "test_data".

- "src" contains five files, some of which you will need to write some code for.
- "test_data" contains the expected outputs for the programs in "src".

1 Activity: Implementing “max.c” and “square.c” [4 points]

While “max.c” and “square.c” have yet to be implemented, it is instructive to take a look at “multiply.c” first, since the other two are to be implemented in much the same manner.

1.1 “multiply.c”, Command Line Arguments, and Standard Input

- Open up the source file “multiply.c” and examine the source code.
- Compile the program and attempt to run it. `multiply` uses a function defined in `library.c`, so you’ll need to include it in the compiler invokation.
- `multiply` expects one integer command line argument. The program waits for something to come in over `stdin`, multiplies it by the provided argument, and then bounces it back to `stdout`.
- Run the program and try it out!
  - Uh oh, looks like we can’t exit the program! Remember, to kill any program, just hit `Ctrl+C` on your keyboard!
  - If we examine the source code, we see that `multiply` is waiting for something called an `EOF`. This is an “end of file” tag, which we will be talking about soon. Basically, it’s an ascii character at the end of every file that tells the computer the file has ended.
- In fact, this program expects input to be routed to `stdin` from a file, hence the `EOF`.
- In Bash environments, we can direct the contents of a file to `stdin` using >.

  ```
  $ ./multiply 10 < ../test_data/multiply_input.txt
  ```

  This can even be used in combination with `stdout` capture using <

  ```
  $ ./multiply 10 < ../test_data/multiply_input.txt > multiply_output.txt
  ```

1.2 “square.c”

Complete the code template provided in “src/square.c”. This program should work in a similar manner to “multiply.c”, and use the `square` function provided in “library.c”. Usage should be as follows:

```
$ ./square inches < input_data.txt
1 inches
4 inches
9 inches
16 inches
```

The second argument “inches” is a units designation, which should be included in the final result.

HINT: If you’re wondering how to convert strings to integers in C, take a look at “multiply.c”!

1.3 “max.c”

Complete the code template provided in “src/max.c”. This program should work in a similar way to “multiply.c” and “square.c”. It must use the `max` function provided in “library.c”. Usage is as follows:

```
$ ./max 0 1 5 2 8 1 7 9 3 8 1
9
```

Each argument after the first (“./max”) is a possible maximum value. Your program need only write the maximum value into `stdout`.

Commit all your files.

2 Activity: Creating a Makefile [10 points]

Now that we have implemented our files, let's create a makefile to compile them automatically! For reference in this section, check the Topic 9 slides. Students who submit makefiles which skip anything listed below aren't going to get full marks!

1. First, create a file called "Makefile" in your "1XC3_L07" directory (i.e., one level up from the source files). Open it in emacs, er... I mean... open it in your favourite text editor.
2. Create a rule which compiles "library.c" to an object file.
3. Create rules which compile "multiply.c", "max.c" and "square.c", using the "library.o" object file. Be sure to set up your prerequisites!
   - For now, let's place the executables in the "src/" directory (i.e., the same location as the source files).
   - Figuring out how to use code that's not in the current working directory is left as an exercise for the student.
4. Now let's add some variables! It's highly advisable to take some time between these steps to make sure your makefile is still working.
   (a) `CC` → The compiler to be used for compilation.
      - Set the value of this variable to `gcc`
      - Replace all occurances of `gcc` in the makefile with an appropriate invokation of the variable.

   (b) `CFLAGS` → The flags with which we want all compilation processes to run.
      - Set the value of this variable to `-Wall`
      - Add it to all compiler invocations.

   (c) `BUILD_DIR` → The name of a directory into which all executables and object code should be placed.
      - Set the value of this variable to `build`.
      - Modify your rules so that any object files or executables which the program produces are placed inside the specified build directory.
      - If such a directory does not exist, create it first!
        – You'll run into difficulty if you try to do this in the makefile itself. Instead, write a short Bash script which performs the above function, and invoke it as part of the recipe!
      - Also remember, you are relocating library.o, so that will need to be reflected in your recipes.

   (d) `SRC_DIR` → The directory containing the source code files.
      - Set the value to `src`.
      - Anywhere you hardcoded the source directory name, replace it with this variable.

   (e) `$@` → Automatic variable designating the target name.
      - Wherever a rule uses the target name in a recipe, replace it with `$@`

   (f) `$<` → Automatic variable designating the first prerequisite.
   `$^` → Automatic variable designating the full list of prerequisites.
      - Use the above two automatic variables to further refine your recipes. Each recipe should use one or the other of these variables.
5. Now add the following additonal rules:
   - `all`
     – This rule should build all final targets (i.e., `max`, `multiply` and `square`).
   - `clean`
     – This rule should delete the build directory and everything in it. Use the `-f` flag to suppress `rm` complaining about missing files or directories.
6. Double check that your makefile builds everything correctly. Create a copy of the file called "part2_makefile", and commit it.

3 Activity: Automated Testing [5 points]

In this section, we will use the provided testing data to automatically run tests on our executables.

1. Make a new rule in your makefile, with a target called `test`. Its prerequisite should be `all`.
2. The first command this rule should execute is to create a "test" directory. This directory will contain all the intermediate data we generate during testing.
3. The final command will be to delete this directory and everything in it.
4. A line should be added to the clean recipe that also deletes the "test" directory. A failed test or command will stop the test rule executing, so it's useful to be able to clean it up too with our pre-established cleanup routine.

3.1 Testing Procedure for "square.c"

- The square executable should be run with the command-line argument "inches", with the "test_data/square_input.txt" file redirected to `stdin`.
- For the above command, `stdout` should be redirected to "test/square_output.txt".
- Finally, use the `diff` command to compare "test_data/square_expected.txt" to "test/square_output.txt".

3.2 Testing Procedure for "multiply.c"

- The square executable should be run with the command-line argument "2", with the "test_data/multiply_input.txt" file redirected to `stdin`.
- For the above command, `stdout` should be redirected to "test/multiply_output.txt".
- Finally, use the `diff` command to compare "test_data/multiply_expected.txt" to "test/multiply_output.txt".

3.3 Testing Procedure for "max.c"

- The square executable should be run with the command-line arguments "4 3 2 1 5 7 8 10 6", in that order.
- For the above command, `stdout` should be redirected to "test/max_output.txt".
- Finally, use the `diff` command to compare "test_data/max_expected.txt" to "test/max_output.txt"

Commit everything and you're finished! Congratulations!