Programming for Engineers Portfolio : Quarter 3, Set 1

These exercises are intended to directly follow the tasks from the first lab of Q3. If you haven't done the lab, you are strongly reccommended to do so first, as it shows how to do all the tasks, and also puts it in a more realistic context: lab1 spec

These exercises are very artificial and simply require you to demonstrate that you can complete the same steps on an unseen code-base. The lab and lectures try to give context as to why these are useful, but here they just need to be done.

1 - Pull source code from remote repository

Clone this repository (elec40004-2019-q3-portfolio-ytc19-master) into a local repository. The fact that you complete the rest of the steps shows that you have completed this step.

2 - Create test infrastructure

Create a test script called `test.sh`. This script should compile programs `p1.cpp` and `p2.cpp`, run them, and then collect the output in `p1.out.txt` and `p2.out.txt`. It should then compare them against pre-generated reference outputs `p1.ref.txt` and `p2.ref.txt`. If any step fails, the script should fail.

3 - Capture the state of the repository using a commit

Add the files `test.sh`, `p1.ref.txt` and `p2.ref.txt` to source control, and create a commit containing the string `s1E3PreReFactor`. Your commits should appear under your imperial login and email address.

4 - Convert the D type into a struct with member functions

Modify `D.hpp` so that it is a struct with public member functions and public member variables. You should remove the `D_` prefix from functions.

5 - Converting uses of the type via a pointer

Convert the program `p1.cpp` so that it compiles against the modified `D` class.

6 - Converting a program that uses a local variable

Convert the program `p2.cpp` so that it compiles against the modified `D` class.

You should retain the direct manipulation of member variables to demonstrate the ability to refactor progressively while maintaining a working codebase. The idea is to incrementally improve and adjust in a disciplined way, rather than trying to do everything at once and ending up in chaotic intermediate repository states that don't compile or work.

7 - Commit and push changes

Ensure that `test.sh` passes, and commit the updated source files using a message containing the string `s1E7PlainClass`, then push to the repository at `https://github.com/ELEC40004/elec40004-2019-q3-portfolio-ytc19-private`.

8 - Convert to a class and apply access modifiers to protect member variables

Convert `D` from a struct to a class, and use access modifiers to ensure that member variables are private, while methods remain public.

9 - Fix p2

Update `p2.cpp` so that it retains the same behaviour, while only using the existing public member functions of `D`.

10 - Commit and push the re-factored code

Ensure that `test.sh` passes, create a commit containing the string `s1E10RefactorComplete`, and then push to your private repo at: `https://github.com/ELEC40004/elec40004-`

Checking and backing up your code

You should check your code as in the lab:

1. Clone it to a completely new directory
2. Run `test.sh` and check it works.

You may also wish to check out the two other commits (`s1E3PreReFactor` and `s1E3PreReFactor`) in the history:

1. Use `git log`, and look for the commit you want to try.
2. Use `git checkout HASH`, where hash is the hexadecimal string associated with the commit you're interested in.
3. Your local repo will now be in the state it was in when you commited, so you can check what state the files are in.
4. If you want to get back to the latest (newest) commit, then you can either look for the most recent hash using `git log` and check it out, or use `git checkout master` to get the most recent commit on the `master` branch (usually the default).

Your private repository acts as a pretty good backup: it is stored off-site in a highly redundant and distributed fashion. The only way it can get damaged is through user error. However, given you are learning, you are encouraged to also maintain some standard `zip` or `tar.gz` style archive backups, just in case. Just make sure that your backup includes the `.git` folder, as that is where the version control meta-data is.

Help, my commit history is messed up!

Part of the goal of the exercise is to make sure that you can modify a code-base in a disciplined way, so the checks for 3, 7, and 10 will be looking at particular commits. However, the assessment scripts will look for the *newest* commits that have those tags. So if you find your git history is a bit messed up (e.g. you didn't commit a particular file), you can effectively copy the original files back into the repo, and then build a new commit history on top of it.

You can also create a completely new local repository, perform the tasks again or copy the files in from the other repo, and make sure the commits line up. You can then add the `--force` parameter to your push command to overwrite the previously pushed version. This is a slightly drastic move, so make sure you have a backup of your repo (including the .git directory) somewhere else, just in case.

For most people, as long as you followed the lab already you know what the steps will be, so it should be straightforward to get the right commits at the right stage if you just work through the instructions in order.