ELECTRICAL AND INFORMATION ENGINEERING
University of the Witwatersrand, Johannesburg
Software Development II – Laboratory 1

An Introduction to Objects and Classes

1 Introduction

The purpose of this laboratory is to introduce object-based programming. This is done in stages, firstly, you are required to simply make use of an existing class/type from the Standard Template Library (STL). Secondly, you need to modify, and enhance, a partially-written `Screen` class. Finally, you are in a position to create your own `Stopwatch` class from scratch.

The main laboratory outcomes are:

1. You are able to construct and use objects including those from the STL as well as programmer-defined classes.
2. You are able to modify and create new member functions for an existing class as well as create your own classes.
3. You consider class design with regard to the suitability of member functions.
4. You consider the differences between a class's interface and its implementation.
5. You are able to create a clean commit history for your solutions and submit them via a GitHub pull request.

1.1 Git and GitHub

Throughout this course we will be making use of Git — a version control system — and GitHub which hosts Git repositories and allows developers to collaborate. Using Git takes practice and patience but it is a critical skill for software developers to have. Git-related instructions are given throughout the lab. It is important that you follow these to the letter as you will only be awarded the lab mark if you have used Git and GitHub correctly. If you do make a mistake, refer to the Git/GitHub Guide for how to correct it and/or ask us for assistance.

An overview of the process is as follows:

1. Commit the starter code to the *master* branch and then create branches for each section of the lab as well as a *solutions* branch to combine all of the exercises.
2. Share all of these branches with your lab partner via GitHub.
3. Check out the section branch for the section that you are going to work on.
4. Complete each exercise and then commit it to this branch.
5. On completion of the entire section, merge the section branch into your local *solutions* branch.
6. Merge the local *solutions* branches of each group member so that all solutions reside in a single *solutions* branch on GitHub.

7. Lastly, make a pull request on GitHub from *solutions* into *master*.

These steps will be explained in detail throughout the lab.

1.1.1 Initializing the Repo

Before starting the lab it is necessary to create a local Git repository, or repo, for tracking the source code. Make sure you have installed Git Bash and Visual Studio Code according to the instructions given in the Resources|Software and Installation Guides section of the course website.

Now download the lab's starter code from the Labs section of the website and extract it to a folder, say `elen3009-lab1`. Open up Git Bash at this location, by right clicking on this directory in File Explorer and selecting "Git Bash Here". Alternatively, use the built-in Git Bash terminal in VS Code. Now type: `git init` to initialise a new repository. Change your Windows settings to show hidden files and you will notice that a `.git` directory has been created in the `elen3009-lab1` directory. The files in this directory (which you should never modify) record and track every change that happens to your source code files.

You should also notice a `.gitignore` file. If you open up this file in text editor (such as Notepad or your IDE) you will see that it consists of a list of folders and file extensions. We are telling Git to ignore these folders, and these types of files, for this repository. In other words, we are not placing them under version control. We typically want to ignore products of the compile/build process such as object files, and the actual executable. These will be regenerated each time we compile the source code; therefore, there is no need to keep track of them. Often, teams also choose to ignore any IDE-related files as team members may be using different IDEs.

Type `git status` and notice, firstly, that you are on the *master* or main branch. Secondly, there is a collection of untracked files and folders shown in red.

We now need to add the untracked files and directories to the *staging area*, also known as the *index*, and then commit them in order to record the initial snapshot of our solution. To stage all the new files, type (note the period): `git add .`

This will add every single file in the repo directory as well as any sub-directories to the staging area (except those ignored by `gitignore`). `git status` will show you that the files are ready to be committed. To commit the files and specify a commit message, type: `git commit -m "Initial commit"`

Typing `git status` now will show you that there is nothing to commit. That is, the state of your working directory is identical to the most recent snapshot in the repository history.

1.1.2 Creating Branches

Using Git typically involves branching to add a new feature, fix a bug, and so on. We will create a branch for each section of the lab and then commit the solutions for the exercises to the relevant branch. To start off, create a branch for each section containing exercises by typing the following:

```
git branch section-2
git branch section-3
git branch section-4
git branch section-5
```

Lastly, we need a *solutions* branch for combining all of our solutions into a single branch, so type: `git branch solutions`

Typing `git status` should show you that you are still on the *master* branch.

1.1.3 Sharing the Repo

In order to share the starter code, which is contained in the "Initial commit", as well as the *section* and *solutions* branches with your lab partner you need to associate your local repository with a (shared) remote repository on GitHub. You will then be able to synchronize your work between the two.

If you type `git remote show` in order to show the remote repos that this local repo is associated with, nothing should be returned because, at this point, the local repo is not associated with any remotes. To add your remote repo on GitHub, login to GitHub and visit the following URL: https://github.com/witseie-elen3009.

You should see a repo for this lab that we have created. Click on the name and copy the **SSH** URL from the *Quick setup* section. Now in Git Bash, type: `git remote add origin <paste SSH url here>`
This associates the remote GitHub repo (called *origin* by convention) with the local repo. To confirm, type: `git remote show`, and *origin* should appear.

The next step is push your local branches to the remote. Do this by typing:
`git push --all origin -u`

The `--all` option says that we want to push all branches. Remember that you have multiple local branches. The `-u` option tells Git that the local branches must now track the upstream or *origin*'s branches. This means that in future you can simply type `git push` or `git pull` without specifying a branch name. For example, if you push/pull while on your *solutions* branch your work will be pushed to/pulled from *origin*'s *solutions* branch.

Now that the starter code and branches are available in the shared GitHub repo, your lab partner needs to *clone* this repo thereby downloading a copy of it onto his/her computer.

On your partner's machine, clone the remote repo using:
`git clone <paste SSH url from GitHub here>`
This is a once-off operation to copy the repo to a new place.

Typing `git status` in the cloned repo will show you that you are also on the *master* branch. To check all the branches that exist in the repo, type `git branch -a`

At this point, each group member should have a copy of the lab 1 repo along with the starter code and the various branches for committing the solutions.

1.1.4 Working on Different Lab Sections

You now need to decide the sections that each of you will work on. The requirement is to alternate sections so, for this lab, one group member will commit the solutions for all of the exercises in Section 2 and Section 4, and the other group member will commit the solutions for the exercises in Section 3 and Section 5.

Although, you are required to alternate sections with regards to the submission of the lab *it is expected that you understand all of the work covered in the lab*. If you are able to, then try to

collaborate closely on all sections. If you cannot do this, you can always work individually on the exercises that you are not going to actually submit — just create a local branch from *master* on which to do the work, which you do not share on GitHub. For example, suppose my partner is going to commit the exercises for Section 3, I could also work on Section 3 in my own local branch by typing: `git branch section-3-steve` after pushing all of the branches as shown in Section 1.1.3.

Before starting any section, you need to change to the correct branch. For example, if you are going to be working on section 2, then you need to type: `git checkout section-2`. You can confirm that you have indeed changed branches by typing: `git status` or `git log`. You are now in a position to start working on the lab exercises in your particular section.

As these exercises are all quite simple in terms of file structure you can setup VS Code in each src folder as shown in the Hello World tutorial that is linked to on the course website. Make sure to commit after each exercise as described below.

Further Git/GitHub instructions will be identified using the sidebar on the left. Detailed instructions will be given for the first two sections of the lab. The remaining sections of the lab need to be completed in the same way. At the end of the lab (Section 6) instructions are given for merging the *solutions* branches and submitting the lab via a pull request on GitHub.

2 C++ Revision

Exercise 2.1

This exercise is provided to help you revise basic C++ concepts that have been covered in Software Development I. It is advisable that you review your notes from that course before attempting the exercise. Provide your solution to this exercise in the `game.cpp` file within the `guessing-game` folder.

Write a program which gives the user five chances to guess a secret random number (between 1 and 100). If they guess the number they win the game and it ends; otherwise they lose after five guesses. After each guess an appropriate message is to be shown: "Guess higher", "Guess lower", "You win" or "You lose".

To generate a pseudo-random number in C++ use the `rand()` function which is part of the `cstdlib` header file. `rand()` generates a random number between 0 and the constant `RAND_MAX` (inclusive). How can you determine the value of `RAND_MAX`? To scale the random number to the right range make use of the modulus or "remainder from division" (`%`) operator.

The `rand()` function generates *pseudo*-random numbers because it returns a number from a pre-determined sequence of random numbers. Each subsequent call returns the next number in the sequence. The *seed* for the random number generator determines the position on sequence from which to start returning numbers. If the seed is identical each time the program is run then the starting point of the sequence remains the same, and the same random numbers are returned.

In order to generate a completely new sequence of numbers each time the program is run it is necessary to seed the random number generator with a unique seed for each run. One

way of doing this is to seed the generator with a number based on the current time using the following code: `srand(time(0));` The `time` function returns the current calendar time in seconds and is part of the `ctime` library. Note that `srand` need only be called once in a program to have the desired randomising effect.

Firstly, make sure that you are on the *section-2* branch. Now take a snapshot of your code by typing:
`git add guessing-game/game.cpp` to add the `game.cpp` file to the staging area, and
`git commit -m "Exercise 2.1"` to record the snapshot.

You can now return to the code as it was at this point in time, at any point in the future, by checking out this particular commit. To see the history of this branch so far, type: `git log`

You should see two commits on the *section-2* branch — the one containing the initial files and the one that you just made. A useful, more compact form of the log can be obtained using: `git log --oneline`

Work through all of the lab exercises in this fashion, committing your changes after completing each exercise. The commit message for each exercise must have the following form: `git commit -m "Exercise <number of the exercise>"`

You should never have multiple versions of a file. This defeats the point of using a version control system. Rather, you should always be *editing the same file or files* as you work through each exercise in a section. Creating a commit history allows you to revisit the state of the files at any point that you committed them. Use `git status` and `git log` often to make sure that you are doing things correctly.

This section is now complete as it only contains a single exercise. You now need to *merge* your commits on the *section-2* branch into the *solutions* branch. To do this you need to first checkout the *solutions* branch because when you do a merge you always *pull the code into the branch that you are currently on*. So, do the following:

`git checkout solutions` to make *solutions* your current branch
`git merge section-2` to merge in the commits from the section branch

This merge is known as a *fast-forward* merge because your *solutions* branch has *not* "diverged" from the *section-2* branch; it is only one commit behind. As a result, no merge commit will be produced. If two branches have diverged it means that each branch contains commits that the other branch does not have. Here, however, the *solutions* branch only contains the "Initial commit" and the *section-2* branch *also contains* this commit along with an additional commit containing the solution to the exercise.

3 The `complex` Type

The STL includes a *template* class which represents complex numbers. More details on this class can be found at https://en.cppreference.com/w/cpp/numeric/complex. Examine Listing 1 to see how objects of this class are constructed and used.

```
1  #include <iostream>  // contains the definition of cout, endl
2  #include <complex>   // contains the complex class definition
3
4  using namespace std; // cout, endl, complex are all part of this
       namespace
5
6  int main()
7  {
8     auto num1 = complex<float>{2.0, 2.0};  // use auto for type
          deduction
9     auto num2 = complex<float>{4.0, -2.0}; // use uniform
          initialisation syntax (curly braces)
10
11    auto answer = num1 * num2; // type deduced for 'answer' is:
          complex<float>
12
13    cout << "The answer is: " << answer << endl;
14    cout << "Or in a more familiar form: " << answer.real()
15         << " + " << answer.imag() << "j"
16         << endl  << endl;
17
18    // answer++;
```

Listing 1: Complex Numbers

A template class usually requires the programmer to supply a type for the class. In the case of the `complex` class, the type supplied is the type that is used for storing the real and imaginary parts of the complex number. In Listing 1 `complex` is instantiated with the `float` type to creating floating point complex numbers. Alternatively, we could have used another numeric type such as `int` or `double`.

Exercise 3.1

Why do you think the commented line of code does not work? Supply your answer as a comment in the source code.

Firstly, make sure that you are on the *section-3* branch. Then commit the solution to this exercise:
`git add .` when you use a '.' this adds *all* the files to the repo
`git commit -m "Exercise 3.1"`

This can be achieved in one command for files that are already being tracked (i.e. files that have already been added to the repo in an earlier commit):
`git commit -am "Exercise 3.1"`

Exercise 3.2

We can improve the readability of the above code slightly by using an *alias* for the rather unwieldy: `complex<float>`. Read up on the `using` keyword and rewrite Listing 1 using a more readable name for `complex<float>`.

Again, do not forget to commit this exercise: `git commit -am "Exercise 3.2"`

Exercise 3.3

The solution of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

is given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write a program, in `quadratic.cpp`, that requests the integers a, b and c from the user, calculates the roots and displays the result. The program must then ask the user if they wish to do another calculation and repeatedly calculate roots for different sets of constants until the user presses "q" to quit. You are required to determine the roots even if they happen to be imaginary.

Hint: a good way of solving this problem is to leverage the functionality of the `complex` type by making extensive use of it, rather than reverting to using primitive types (`float` etc).

You may assume that the user will not make any input errors such as supplying a character when a number is required. This is not really a reasonable assumption but we will only cover error checking later in the course.

This is the *final reminder* to commit each exercise, when you complete it, to the correct branch and to merge these branches into *solutions* when the entire section is complete. For above exercise, type: `git commit -am "Exercise 3.3"`

Having completed this section, you now need to *merge* your commits on the *section-3* branch into the *solutions* branch. Remember, that the *solutions* branch that is referred to here is a local branch belonging to the repo of the group member doing the exercises in section 3. The *solutions* branch referred to in Section 2 is also a local branch belonging to the repo of the group member who elected to do the exercises in Section 2. At the moment these branches are entirely independent of each other. At the end of the lab, the changes to each local *solutions* branch will be shared, and combined, via GitHub.

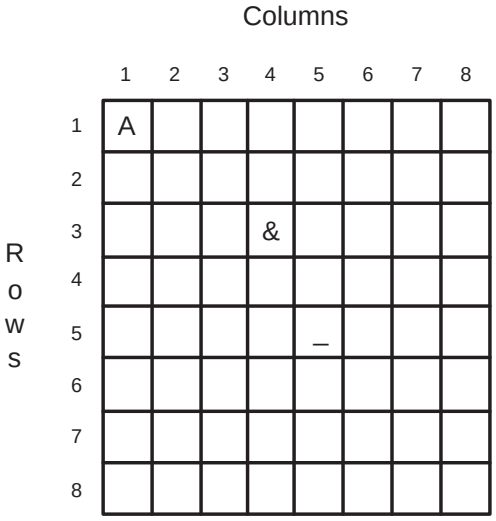
To merge in the *section-3* branch, first checkout the *solutions* branch because when you do a merge you always *pull the code into the branch that you are currently on*. Type the following:

```
git checkout solutions
git merge section-`\thesection`
```

This merge is also a *fast-forward* merge because the *solutions* branch, which only contains the "Initial commit", is simply a couple of commits behind the *section-3* branch. As a result, no merge commit will be produced.

4 Modelling a `Screen`

Consider the concept of a screen which is composed of text characters.



The screen consists of a grid. Each position in the grid is indexed by specifying the appropriate row and column. For example, "A" is at position (1,1), while the ampersand is at position (3,4). A cursor is used to identify the position on the screen that will next be read from or written to. So in the diagram above, the next character to be written to the screen will be placed at position (5,5). Note that the cursor is never actually displayed.

This concept or abstraction is captured by the `Screen` class. The screen module (`screen.h` and `screen.cpp`) as well as a `main` function which exercises a `Screen` object are included in the source code for this lab.

Exercise 4.1

Before starting this exercise, note that we are now dealing with multiple C++ files and not a single file as in previous exercises. In order for Visual Studio to compile all these files and link them together it is necessary to edit the `tasks.json` file so that the correct arguments are passed to the compiler. Make sure that the `args` value is as follows:

```
"args": [
  "-fdiagnostics-color=always",
  "-g",
  "${workspaceFolder}/**.cpp",
  "-o",
  "${fileDirname}\\${fileBasenameNoExtension}.exe"
],
```

Once you have done this compile and run the main program which demonstrates the capabilities of a `Screen` object. Relate the output that is generated, to the source code of the `main` function and `screen.h`.

Resisting the temptation to examine `screen.cpp`, use the member functions declared in `screen.h` to generate a 6×6 screen that contains your first initial. For example, if your initial is "S" then you could generate the following:

```
******
*
*
******
     *
******
```

Exercise 4.2

Now open `screen.cpp` and try to understand the source code of the various member functions. `Screen` is implemented using the `string` class so it will be helpful to review the reference sheet for the `string` class (on the course web site) or the online documentation.

Identify *three* different situations in which the `const` keyword is being used in `screen.h`, and explain the meaning of `const` in each of these situations. Give your answer using comments in the source code.

Also, investigate and explain the meaning of `string` class's `at` member function (used in `Screen::reSize`). Give your answer as a source code comment.

Exercise 4.3

We wish to provide an additional `move` function which accepts a "direction". The direction can take on one of the following values:

- HOME
- FORWARD
- BACK
- UP
- DOWN
- END

Overload the existing `move` function with a function having the following declaration:

```
void move(Direction dir);
```

where `Direction` is a scoped or strongly-typed enumeration. Read up on strongly-typed enumerations and prefer to use these over pre-C++11 enumerations.

Note that this function can be implemented entirely in terms of existing functionality, which is what you should do.

> Follow the DRY principle — Don't Repeat Yourself

Is this member function a necessity for clients of `Screen`? Provide your answer as a comment.

Exercise 4.4

The `Screen` functions `forward` and `back` wrap around. For example, if the cursor is positioned at the bottom right-hand position of the screen, a call to `forward` will cause the cursor to advance to the top left-hand position of the screen.

However, the functions `up` and `down` do not wrap around and report errors if used when the cursor is positioned on the top and bottom rows of the screen, respectively. It is important to offer the client of a `Screen` object a consistent interface. "Wrap-around" functionality should be provided for all relative movement functions. Implement this functionality for both `up` and `down`.

Exercise 4.5

We would like the ability to draw empty squares on our screen, like so:

```
XXXX
X  X
X  X
XXXX
```

Create a member function that accepts the co-ordinates of the top-left corner of the square as well as the length of the sides and draws the square on the screen. Ensure that your function provides proper error checking. This function is relatively complex and it may be a good idea to implement it by making use of other `private` helper functions.

Do you need access to the internal representation of the `Screen` class in order to implement this function or can you simply use the existing interface? Does a function like this really form part of the responsibilities of a `Screen` object? Give reasons for your answers.

Exercise 4.6

The internal representation of the `Screen` class as a `string` is perhaps not as intuitive as it could be, resulting in member functions that are not easily understandable. Can you think of a better (more intuitive) internal representation that could be used without the existing `public` interface having to change?

Why is it important to avoid changing the class's interface, and why are we free to change the implementation?

As before, answer with source code comments.

When you attempt to merge *section-4* into the *solutions* branch (which already contains the solutions for *section-2*) a merge commit will be produced. This is because a merge is happening between two divergent branches (each of which contains commits that the other does not have). The merge commit that results contains the union of the code from both of the parent branches (*solutions* and *section-4*). A commit message is required for this merge commit so Git Bash will display the default message ("Merge branch 'section-4' into solutions") in a text file in the editor that you have previously configured. You can accept the default commit message by simply closing the file that pops in your text editor. The merge should then automatically take place, and you can can check this by typing: `git log --oneline`.

5 Modelling a Stopwatch

Exercise 5.1

Model a simple stopwatch, using a class, which times, in seconds, how long a section of code takes to execute. Think carefully about how a stopwatch behaves and write down its responsibilities *before* deciding on its interface and implementation.

Implement the stopwatch in the files `StopWatch.h` and `StopWatch.cpp`, and provide a `main` function which uses your stopwatch. The function `getProcessTime` is given which allows you to obtain the amount of time (in seconds) that has passed since your program started executing. You should make use of this function *within* your class.

When you attempt to merge *section-5* into the *solutions* branch (which already contains the solutions for *section-3*) a merge commit will also be produced. This is because the branches being merged have diverged. Again, accept the default commit message by simply closing the file that pops in your text editor. The merge should then automatically take place, and you can can check this by typing: `git log --oneline`.

6 Submitting the Lab

6.1 Combining Solutions

The lab has now been completed and it is necessary to submit the solutions, via a GitHub pull request, so that the lab can be marked.

At the stage, there are 3 *solutions* branches that exist:

1. There is a remote *solutions* branch on GitHub which was pushed at the start of the lab. This branch only contains the initial commit.
2. One lab partner will have a local *solutions* branch containing the solutions for sections 2 and 4.
3. The other lab partner will have a local *solutions* branch containing the solutions for sections 3 and 5.

In order to combine all the solutions onto the remote *solutions* branch, each lab partner needs to push their local branches to GitHub. The order in which this is done does not matter. To push your local *solutions* branch type `git push` when you are on the branch.

The first push to *solutions* will succeed without a problem. However, when it comes to the second partner pushing, Git is going to complain because the local *solutions* branch and the remote *solutions* branch contain different work. To resolve this, you need to first merge the remote *solutions* branch into your local *solutions* branch. To do this, the second partner who attempted to push needs to type: `git pull` to pull the remote *solutions* branch into their local branch. Again, a merge commit will created representing the union of all of the commits on the local and remote *solutions* branches. Do a `git log` to ensure that all of the solutions are now present on the *solutions* branch. The next-to-last step is to *git push* the branch back up to GitHub. The last step should be for the other lab partner to pull down the remote *solutions* branch so that both partners have a copy of all of the solutions.

In general, the process to be followed when sharing code on a branch is as follows:

1. First pull down the remote version of the branch from GitHub so that you receive any updates that your collaborators have made.
2. Merge their changes into your local branch.
3. Push your updated local branch back to GitHub to make your changes available to everyone.

6.2 Creating a Pull Request on GitHub

The final step is to make a pull request on GitHub to pull the work on your remote *solutions* branch into your remote *master* branch. Refer to the Git/GitHub guide for how do this.

6.3 Lab Evaluation

The group will only be awarded the mark for the lab if it is submitted correctly (refer to Figure 1). This means that:

- The pull request for your submission must appear when using the search criteria given in the Git/GitHub guide.
- The solutions to all exercises must be present in the pull request. There must be no missing or partial solutions.
- Each group member must always commit under their GitHub username so that their profile picture appears next to their username. If more than one commit is made which is not linked to a member's GitHub account then that member will not receive the lab mark. See here for more information.
- Exercise commits must be named as follows: "Exercise <number of the exercise>"
- The solution to each exercise must be contained within a single commit.
- Each exercise commit must only include the code modifications required for that particular exercise. There must be no modifications related to any other exercises.
- Your commit history must contain appropriate merge commits.

If one group member does not submit all of their required commits (alternate sections) then they will not receive the mark for the lab.

Be sure to check that your pull request on GitHub is correct!

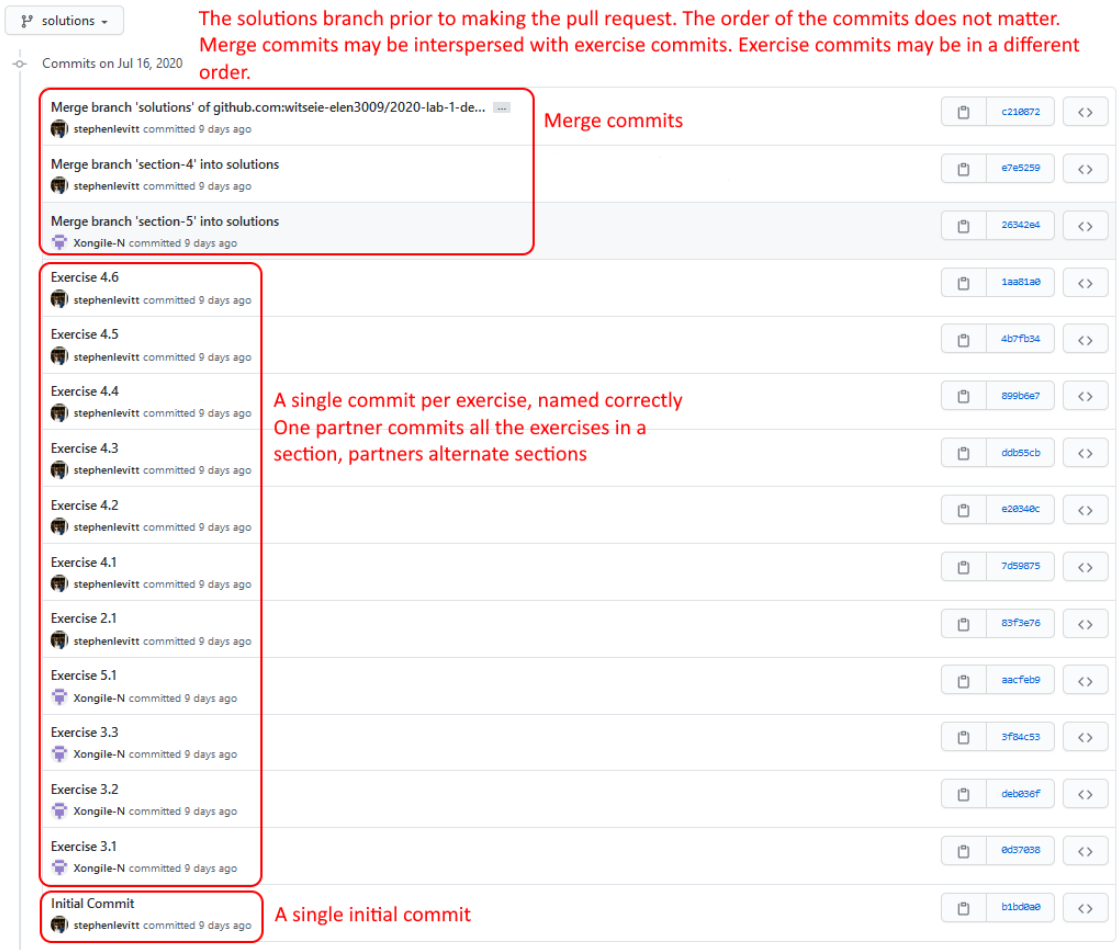
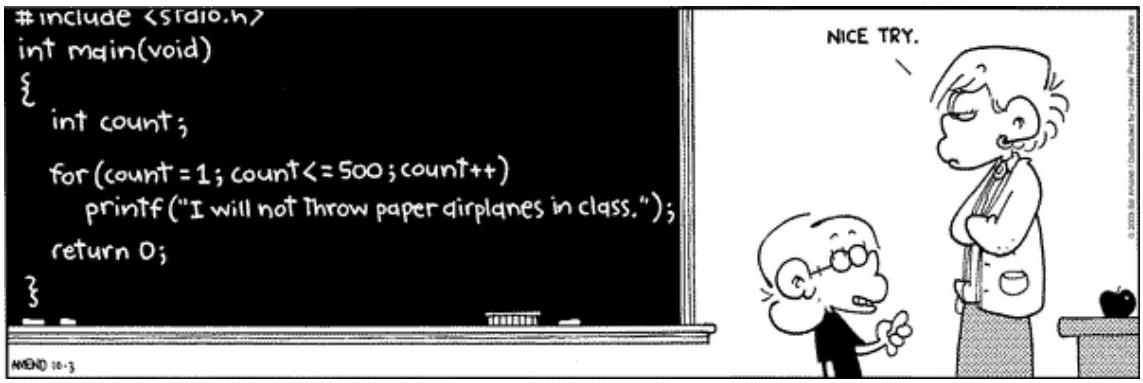
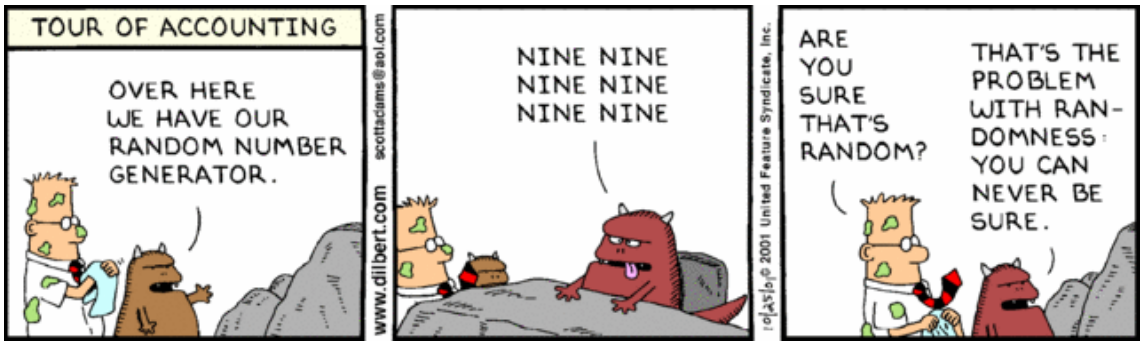


Figure 1: A clean commit history for Lab 1



Source: http://www.foxtrot.com/



Source: http://dilbert.com/strips/comic/2001-10-25/