UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Electrical and Information Engineering
Software Development II – Laboratory 1

An Introduction to Objects and Classes

1 Introduction

The main 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.

In this lab you will also make use of Git – a version control system – and GitHub for versioning and submitting your solutions. Git-related instructions are given in shaded blocks 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 submitted your solutions correctly. If you do make a mistake, refer to the Git/GitHub Guide for how to correct it.

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.

2 C++ Revision

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 according to the instructions given in the Git/GitHub Guide (in the Labs section of the 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 and 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) essentially record and track every change that happens to your source code files.

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*, *working tree* or *index* and then commit them in order to record a snapshot of our project at this particular point in time.To stage all the new files, type (note the period): `git add .` `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.

Using Git typically involves branching to add a new feature, fix a bug, and so on. We will create a branch on which to code the solutions for the exercises, called *solutions*. Do this with the following command: `git branch solutions`

Typing `git status` should show you that you are still on the *master* branch. In order to change to the *solutions* branch, you need to type: `git checkout solutions`

Confirm that you have indeed changed branches by typing: `git status`. You are now in a position to start working on the lab exercises.

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.

Create a new empty C++ file called `guessing-game.cpp` in the `elen3009-lab1` folder. Provide your solution to this exercise in this file.

Write a program which gives the user 5 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 5 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.

Now take a snapshot of your code by typing:
`git add .` to add the new guessing-game.cpp file to the staging area
`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 your project so far, type:

```
git log
```

You should see two commits - 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>"
```

Note, you should *not* be saving the solution to each exercise in its own folder or file. This defeats the point of using a version control system. Rather, you should be editing the same files as you work through each exercise. 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.

3 The `complex` Type

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

```
// complex.cpp
// Multiplying complex numbers

#include <iostream>
#include <complex> // required for the complex class

using namespace std;

int main()
{
    complex<float> num1{ 2.0, 2.0 };  // using C++11 uniform initialisation syntax
    complex<float> num2{ 4.0, -2.0 }; // old syntax: complex<float> num2(4.0,-2.0);

    auto answer = num1 * num2; // using C++11 auto keyword,
                               // answer is of type: complex<float>

    cout << "The answer is: " << answer << endl;
    cout << "Or in a more familiar form: " << answer.real() << " + " <<
        answer.imag() << "j" << endl;

    // answer++;

    return 0;
}
```

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.

Don't forget to commit this solution:

```
git add .
git commit -m "Exercise 3.1"
```

or in one go for files that are already being tracked (i.e. no new files in the repo):

```
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 at `https://docs.microsoft.com/en-us/cpp/cpp/aliases-and-typedefs-cpp` and rewrite Listing 1 using a more readable name for `complex<float>`.

Again, don't forget to commit after each solution. This is the last reminder.

Exercise 3.3

The solution of a quadratic equation of the form

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

is given by

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

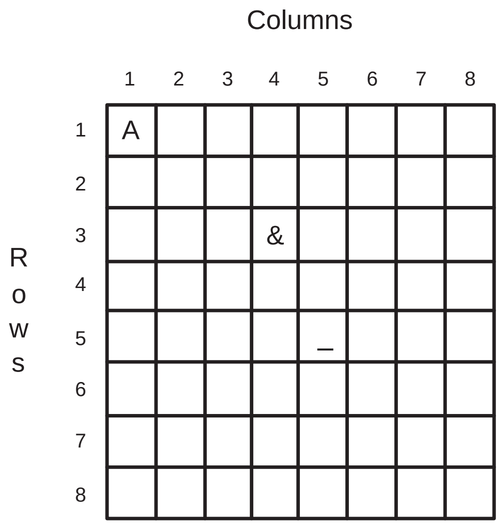
Write a program that requests the integers a, b and c from the user, calculates the roots and displays the result. The program must then ask the user if they wish to do another calculation and repeatedly calculate roots for different sets of constants until the user presses "q" to quit. You are required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

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

Create a project with the above three files 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 at `https://docs.microsoft.com/en-us/cpp/standard-library/basic-string-class`.

Identify *three* different situations in which the `const` keyword is being used, 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`).

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. 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.

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 test it. The function `getProcessTime` is provided 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.

Your solutions now need to be submitted on GitHub via a pull request. If you and your partner have worked independently on the lab, then you must select one of your repositories to submit.

The first step is to ensure that you have a clean commit history. If you have been following the instructions concerning Git correctly, then typing: `git log --oneline` will produce the following:

```
aac7048 (HEAD -> solutions) Exercise 5.1
0fcbd8c Exercise 4.6
f48e972 Exercise 4.5
bfbd3ef Exercise 4.4
32fa7e8 Exercise 4.3
26d8206 Exercise 4.2
8033c75 Exercise 4.1
09a7767 Exercise 3.3
8fb961c Exercise 3.2
5c47174 Exercise 3.1
50aec76 Exercise 2.1
b9700c9 Initial commit
```

Your commit hashes will be different but otherwise the log should look exactly like this. There is a single commit for each exercise which contains the solution for that exercise. The differences between two successive commits should only be the code changes (added/removed/modified lines of code) that were made in order to solve the particular exercise. This implies that for the series of exercises in sections 3 and 4 *you are always working in the same files* as you progress through the exercises and take snapshots. Lastly, all of these commits take place on the *solutions* branch.

If your commit log looks different to this then you need to clean up the commit history *before submitting*. Refer to the Git/GitHub guide (see the Laboratories section of the course website) on how to do this.

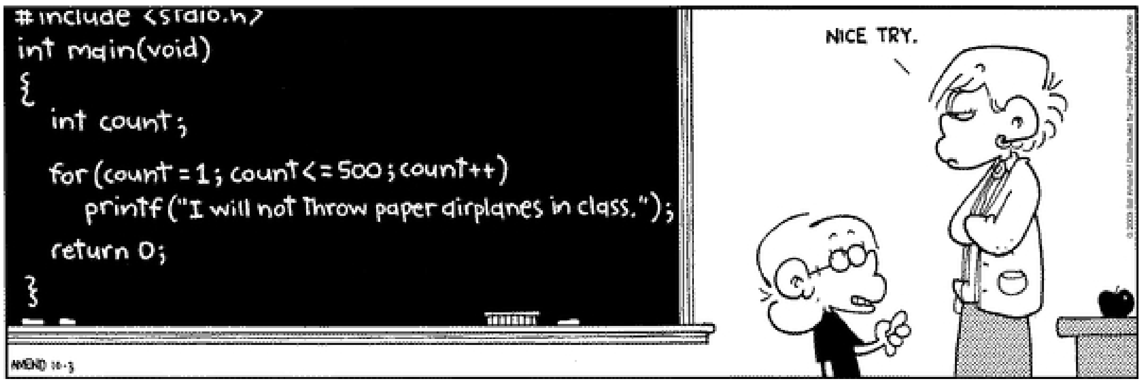
The second step is to associate your local repo with a remote repository on GitHub. You will then be able synchronize your work between the two.

Configure Git Bash with the Wits proxy settings as per the instructions in the Git/GitHub guide and then type the following: `git remote show`. This command shows the remote repos that the 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-elen3009-2017`.
You should see your repo for this lab. Click on the name and copy the **SSH** url (the HTTPS url does not work through the Wits proxy) 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 third 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 two branches, a *solutions* branch as well as a *master* branch. 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.

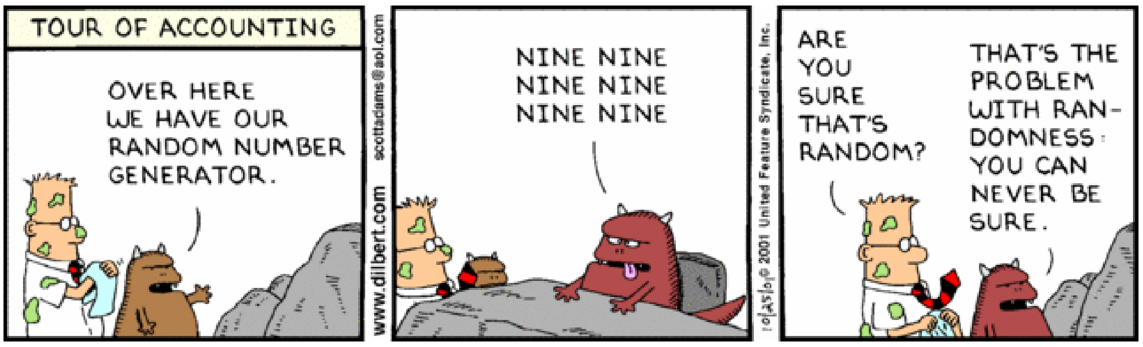
The final step is to make a pull request on GitHub to pull the work on your remote *solutions* branch into your remote *master* branch. Refer to the Git/GitHub guide for how do this.



```
#include <stdio.h>
int main(void)
{
  int count;

  for(count = 1; count<=500; count++)
     printf("I will not Throw paper airplanes in class.");
  return 0;
}
```

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


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RAN-DOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

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