CS264 Laboratory Session 4

John McDonald

20^{st} October 2015

Deadline: All solutions to be submitted by 5pm Tuesday 3^{rd} November 2015.

1 Lab objectives

In this lab you will start to work with classes in C++.

Learning Outcomes
From a C++ perspective you will be able to implement simple C++ classes and use those clases to instatiate and use instances of those classes within your programs.

2 Questions:

For each of the problems given below write a C++ program that provides a solution. Each box provides a filename to use (or in certain cases multiple filenames). Please ensure that you use those filenames. Failure to do so can result in loss of marks.

Step 0.1: For this week's exercises you should create a sub-directory off of the top level `Labs` directory from last week called `Lab4`. **Given that each of the exercises this week require multiple C++ files you should create a separate subdirectory for each. That is, within the `Lab4` directory create `Ex1`, `Ex2`, and `Ex3` subdirectories for the code for each of the respective exercises**. Be sure to commit at least after each exercise with appropriate commit messages.

Remember:

- **Add source files to your git repository and commit changes regularly.**
- **All commits should be accompanied by messages that would allow a lecturer or demonstrator to understand the purpose of the commit.**
- **Comment your code.**
- **Use proper indentation for function and control structures.**

Exercise 1.1: Take your solution to Exercise 1.3 of Lab 2 (i.e. the dice problem) and rewrite it using C++ classes. In particular define a new class called `die`. The class should permit a die of any positive number of sides to be represented. The class should also enforce a constraint that a die should have no less that 4 sides. If the client code tries to construct a die of less than 4 sides the die should default to 6 sided (and print a informational warning message to the standard error).

The class should provide three public methods (not including constructors, *etc.*):

```
int roll();
int getValue();
int getNumsides();
```

Using the resulting `die` class create a program that has the same functionality as required in Exercise 1.3 of Lab2.

You should save the driver source file (i.e. with the `main` function) in a file called `exercise1.cpp`. The code for the `die` class should be stored in separate header and implementation files (i.e. `die.h` and `die.cpp`).

Exercise 1.2: Create a class called `MyComplex` for performing arithmetic with complex numbers. Write a driver program to test your class. Complex numbers have the form $z = a + ib$ where $i = \sqrt{-1}$.

Use double variables to represent the `protected` members of the class. Provide a constructor function that enables an object of this class to be initialised when it is declared. You should also define a default constructor which initialises both a and b to be set equal to zero.

You should also provide *getter* and *setter* methods for both the real and imaginary components of the `MyComplex` number:

```
double getReal()
void setReal(double)
double getImag()
void setImage(double)
```

In this question you should implement the following methods **without** using operator overloading.
Addition of two `MyComplex` numbers, using the following prototype:

```
MyComplex Add(const MyComplex &z)
```

Subtraction of two `MyComplex` numbers, using the following prototype:

```
MyComplex Subtract(const MyComplex &z)
```

Multiplication of two `MyComplex` numbers, using the following prototype:

```
MyComplex Multiply(const MyComplex &z)
```

For example given three objects, `a`, `b`, and `c`, the code

```
c = a.Add(b);
```

should be equivalent to $c = a + b$.

You should also provide a `print` function to output a complex number to the screen in the following format: "2 + 4i". The function should have the following prototype:

```
void print()
```

Your code should include *driver* source code in a file called `exercise2.cpp` that demonstrates each aspect of the `MyComplex` class. The code for the `MyComplex` class should be stored in a separate header and implementation file (i.e. `MyComplex.h` and `MyComplex.cpp`).

Exercise 1.3: For this exercise you should start by copying the source files from the previous exercise into the `Ex3` directory. Starting from this point the objective of this exercise is then to reimpliment the operations from Exercise 1.2 using *operator overloading*. In particular you should provide implementations for the following operators: `+`, `-`, `*`.

Again, your code should include *driver* source code in a file called `exercise3.cpp` that demonstrates each aspect of the `MyComplex` class. The code for the `MyComplex` class should be stored in a separate header and implementation file (i.e. `MyComplex.h` and `MyComplex.cpp`).