CSCI-1200 Data Structures — Fall 2014
Lab 4 — Testing and Debugging

Testing and debugging are important steps in programming. Loosely, you can think of testing as verifying that your program works and debugging as finding and fixing errors once you've discovered it does not. Writing test code is an important (and sometimes tedious) step. Many software libraries have "regression tests" that run automatically to verify that code is behaving the way it should.

Here are four strategies for testing and debugging:

1. When you write a class, write a separate "driver" main function that calls each member function, providing input that produces a known, correct result. Output of the actual result or, better yet, automatic comparison between actual and correct result allows for verifying the correctness of a class and its member functions.

2. Carefully reading the code. In doing so, you must strive to read what the code actually says and does rather than what you think and hope it will do. Although developing this skill isn't necessarily easy, it is important.

3. Using the debugger to (a) step through your program, (b) check the contents of various variables, and (c) locate floating point exceptions and segmentation violations that cause your program to crash.

4. Judicious use of `std::cout` statements to see what the program is actually doing. This is useful for printing the contents of a large data structure or class, especially when it is hard to visualize large objects using the debugger alone.

Points and Rectangles

The programming context for this lab is the problem of determining what 2D points are in what 2D rectangles. For rectangles, we will assume they are aligned with the coordinate axes, as shown in the figure below. This makes it easy to represent and to test if a point is inside. Our code will store points in rectangles and determine which points are in which rectangles. This is a toy example of problems that must be addressed in graphics and robotics.

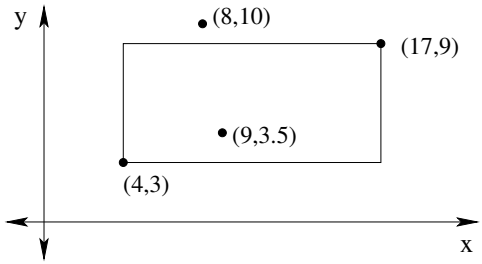

Figure 1: Example of a rectangle aligned with the coordinate axes — the only type of rectangle considered here. The rectangle is specified by its upper right corner point, $(17, 9)$, and its lower left corner point, $(4, 3)$. The point $(9, 3.5)$ is inside the rectangle, whereas the point $(8, 10)$ is outside.

Please download the following 3 files needed for this lab:
http://www.cs.rpi.edu/academics/courses/fall14/csci1200/labs/04_debugging/Point2D.h
http://www.cs.rpi.edu/academics/courses/fall14/csci1200/labs/04_debugging/Rectangle.h
http://www.cs.rpi.edu/academics/courses/fall14/csci1200/labs/04_debugging/Rectangle.cpp

Checkpoint 1

Examine the provided files briefly. `Point2D.h` has a simple, self-contained class for representing point coordinates in two-dimensions. No associated `.cpp` file is needed because all member functions are defined in

the class declaration. `Rectangle.h` and `Rectangle.cpp` contain the start of the Rectangle class. They also contain a bug. Please read the code now to see if you can find it. Do not worry if you can not, ***but do not fix it*** in the code if you do!

Your job in this checkpoint is to complete the implementation of the Rectangle.cpp class. Look through Rectangle.h and Rectangle.cpp to determine what functions need to be added. Then, compile these files and remove any compilation errors (but do not fix runtime logic bugs in the provided code).

Checkpoint 2

Create a new file named `test_rectangle.cpp`. Create a `main` function within this file. In the main function, write code to test *each of the member functions*. For example, write code to create several rectangles, and print their contents right after they are created. Write code that should produce both true and false in the function `is_point_within`. In fact, if there is non-trivial logic in any function, the test code should call the function several (or even many) times with varying inputs to test all the possible branches or paths through the conditionals and loops to ensure every line of code is run at least once. Write code to add points (or not) to a rectangle. Write code to find what points are contained within the bounds of both rectangles.

To complete this checkpoint, show a TA your test cases and the error(s) that those test cases reveal. After doing this you should be able to spot that there is an error in the provided code (as well as, perhaps, errors in your own code). Even if you know where the bug or bugs occur, **please do not fix them yet**.