CSCI-1200 Data Structures — Spring 2016
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.

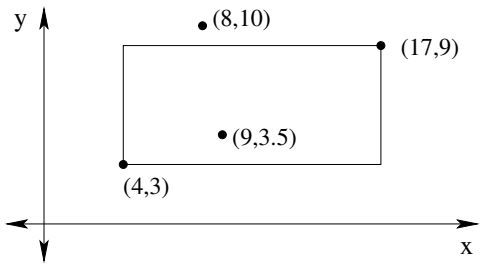

Figure 1: Example of a rectangle aligned with the coordinate axes — the only type of rectangle considered here. The rectangle is specified by its upper right corner point, $(17, 9)$, and its lower left corner point, $(4, 3)$. The point $(9, 3.5)$ is inside the rectangle, whereas the point $(8, 10)$ is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

Please download the following 3 files needed for this lab:
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/labs/04_debugging/Point2D.h
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/labs/04_debugging/Rectangle.h
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/labs/04_debugging/Rectangle.cpp

Checkpoint 1

Examine the provided files briefly. `Point2D.h` defines 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.

We have placed an intentional bug in the Rectangle class! Please read the code now to see if you can find it. Do not worry if you can not. If you do find the bug, ***do not fix it!***

First, 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).

Next, 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**.

Checkpoint 2

Now, we will practice using the debugger to find and fix errors. Today we'll learn how to use the `gcc/g++` command line debugger, `gdb` from the GNU/Linux, MacOSX, or Cygwin terminal. *NOTE: On Mac OSX, you are probably actually using the* `llvm/clang++` *compiler, so you'll use* `lldb` *instead of* `gdb` *in the instructions below.* If you're using Windows and you didn't already install gdb for Cygwin, you'll need to upgrade Cygwin to grab that package. Many introductory `gdb` debugging tutorials can be found on-line; just type `gdb tutorial` into your favorite search engine. Here are a couple:

http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html
http://www.cs.cmu.edu/~gilpin/tutorial/

After you learn and demonstrate to your TA the basics of debugging with `gdb/lldb` you are encouraged to explore other debuggers, for example, the debugger built into your IDE, but we do not provide those instructions. You may also want to try a graphical front end for `gdb`: http://www.gnu.org/software/ddd/ After today's lab you can use your favorite search engine to find basic instructions for other debuggers and figure out how to do each of the steps below in your debugger of choice.

Note that in addition to a standard step-by-step program debugger like `gdb/lldb`, we also recommend the use of a *memory debugger* for programs with *dynamically-allocated memory*, or anytime you have a segmentation fault or other confusing program behavior. Information about memory debuggers is available here:

http://www.cs.rpi.edu/academics/courses/spring16/csci1200/memory_debugging.php

After today's lab, you should be comfortable with the basics of debugging within your preferred development environment. Keep practicing with the debugger on your future homeworks, and be prepared to demonstrate debugger skills when you ask for help in office hours.

1. Run your program that has tests for the basic member functions. Even though the program will compile and run, it will not give the correct output. You may be suspicious about a place in your code where the error occurs. It is time to start the debugger.

2. **Getting started:** When you plan to use the `gdb` (or `lldb`) debugger to investigate your program you will need to build your executable differently. Specifically you need to *link* the code with the `-g` option to add *debug information* (including, source code line numbers!) to the executable. For example, to compile & link all of your files for this lab type:

   ```
   g++ -Wall -g Rectangle.cpp rectangle_test.cpp -o rect_test
   ```

   In order to start `gdb` type

   ```
   gdb rect_test
   ```

   This puts you into the command-line debugger. Type `help` in order to see the list of commands.

There are several commands for setting breakpoints. You can set a breakpoint by specifying a function name or a line number within a file. For example

```
break main
```

sets a breakpoint at the start of the main function. You can also set a breakpoint at the start of a member function, as in

```
break Rectangle::add_point
```

Finally, to set a breakpoint at a specific line number in a file, you may type:

```
break foo.cpp:65
```

Set a breakpoint at some point in your code just before (in order of execution!) you think the first error might occur. Finally, in order to actually start running the program under control of the debugger, you will need to type `run` at the `gdb` command line.

3. **Stepping through the program:** You can step through the code using the commands `next` (move to the next line of code), `step` (enter the function), and `finish` (leave the function). The command `continue` allows you to move to the next breakpoint.

4. **Examining the content of variables:** You can use `print` to see the values of variables and expressions. You can use the command `display` to see the contents of a particular variable or expression when the program is stopped.

5. **Program flow:** The command `backtrace` can be used to show the contents of the call stack. This is particularly important if your program crashes. Unfortunately, the crash often occurs inside C++ library code. Therefore, when you look at the call stack the first few functions listed may not be your code. Find the function on the stack that is the nearest to the top of the stack that is actually *your code*. By typing `frame N`, where `N` is the index of the function on the stack, you can examine your code and variables and you can see the line number in your code that caused the crash. Type `info locals` and `info args` to examine the data in the function. Type `list` to see the source code.

6. **Breakpoint on Variable Change:** The last powerful debugger feature you will find out about today is variable monitoring. Create an instance of a `Point2D` object called `pt` in your main function and change its coordinates using the `set` member function of `Point2D`. You will now monitor the change of this point using the debugger.

   You can use the command `watch` to halt the program when a variable or expression changes. For example, type `watch p.m_x`. You can try the the `rwatch` command which allows you to break on a read of the value (not just a change).

Continue to experiment with these different debugger operations and **only when you feel comfortable showing off all these features put your name in the queue for checkoff for this checkpoint.** The TA/mentor will be asking you to demonstrate these features, especially how you can make the debugger pause when a variable is changed. You will also be asked questions about the other steps in debugging.

Checkpoint 3

Programmers are often called on to debug other programmer's code. It can be a frustrating experience if the code is not clearly written.

Please download this additional file:
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/labs/04_debugging/buggy_test_rectangle.cpp

Compile this file along with your corrected versions of `Points2D.h`, `Rectangle.h`, and `Rectangle.cpp`. It should compile without errors. If you are using clang++ on Cygwin, you may need to add the option `-fno-exceptions` option to the compile command as follows:

```
clang++ -Wall -g -fno-exceptions Rectangle.cpp buggy_rectangle_test.cpp -o buggy_rect_test
```

Run the program. If your corrections for checkpoints 1 & 2 are correct, this buggy test program should produce a segmentation fault. Run the compiled program with gdb. Using backtrace, locate and correct the error causing the segmentation fault. The problem will probably not be at position #0 in the backtrace, but rather somewhere higher in the stack.

To complete this checkpoint and the entire lab, correct the source of the segmentation fault, and be prepared to demonstrate how you found the error.