CSCI-1200 Data Structures — Fall 2012
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 `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 Figure 1. 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.

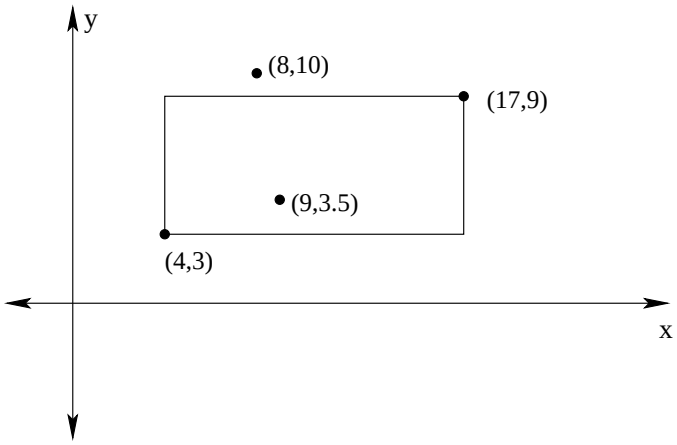

Figure 1: Example of a rectangle aligned with the coordinate axes — the only type of rectangle considered here. The rectangle is specified by its upper right corner point, $(17, 9)$, and its lower left corner point, $(4, 3)$. The point $(9, 3.5)$ is inside the rectangle, whereas the point $(8, 10)$ is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

Please download the following 3 files needed for this lab:

```
http://www.cs.rpi.edu/academics/courses/fall12/ds/labs/04_debugging/Point2D.h
http://www.cs.rpi.edu/academics/courses/fall12/ds/labs/04_debugging/Rectangle.h
http://www.cs.rpi.edu/academics/courses/fall12/ds/labs/04_debugging/Rectangle.cpp
```

Checkpoint 1

Start by creating a project and adding the files `Point2D.h`, `Rectangle.h`, and `Rectangle.cpp`. Examine these 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 to 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 (within Visual Studio put this within the current project) and call it `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 a function, the test code should call the function several (or even many) times with varying inputs to test all the possible conditions.) Write code to add points (or not) to a rectangle. Write code to find what points are contained in 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 3

Now, we need to practice using the debugger to find and fix errors. Visual Studio has an associated visual debugger. If you compile with g++ on `cygwin`, you can use the separate command-line debugger `gdb`. Other debuggers are available, many of them built on top of `gdb`. Of special note, `gdb` may be run from inside of the `emacs` and `xemacs` editors.

In the following outline, Step 1 is for everyone. Step 2 shows you how to get started, and has separate instructions for using the Visual Studio debugger and using cygwin/g++/gdb. Steps 3 and beyond are written primarily for Visual Studio. Those of you using other debuggers should read the Visual Studio instructions and then adapt them to your debugger. Specific pointers to `gdb` commands are provided at the end of each item.

Many introductory `gdb` debugging tutorials can be found on-line; just type `gdb tutorial` into Google. Here are three:

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

You may re-enable your internet connection to read reference material specific to your debugger & development environment.

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:

   - **Getting started with the Visual Studio debugger:** First, make sure you are in *debug mode*, so that you can follow the instructions below. Begin by *Setting a breakpoint.* In the source code, go to the file where the error might have originated. Choose a line to start on, right click anywhere on that line of code, and go to `Breakpoint -> Insert Breakpoint`.

     Now, look at the main menu and click `Debug -> Start` (F5). This will start your program under control of the debugger. A console display will immediately pop up. Your program will execute until reaching the breakpoint you've set.

     At this point, several windows will appear including the source code window. You should still be able to see the console where you typed the input, but this may be hidden. It is important to look back and forth between this and the .net display.

   - **Getting started with g++/gdb:** If you are using `gdb` or any other debugger that works with code compiled by `g++`, you will need to compile your code differently. In particular, you will need to compile using the `-g` option, as in

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

This creates code with debugging information stored about the variables and functions in the program. (Options that you set with Visual Studio compiler generate this debugging information automatically.) Note that the format of this information is different for different compilers, so code compiled using `g++` can not be debugged using Visual Studio.

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 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
```

to set a breakpoint at line 65 of file `foo.cpp`. 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.

As mentioned above, the remaining instructions are centered on the use of Visual Studio. Each ends, however, with pointers to equivalent `gdb` commands. Use the tutorials to figure out the differences if you are using `gdb`.

3. **Stepping through the program:** You can now step through your program one line at a time. Use F10 to step to the next line of code in the current function. This executes this line before stopping and redisplaying (it happens very quickly, though). This works even if the line involves a function call. If you want to see what happens inside the function call, typing F11 will put you into the called function's code. Try not to do this at calls to standard library functions. It can get messy. If you do, Shift-F11 will get you out.

   Hit F10 several times and watch the console display. The output lines of code you wrote will appear so that you can see what's happening there.

   In **gdb** you can step through the code using the commands `next` and `step`. The command `continue` allows you to move to the next breakpoint.

4. **Content of variables:** The default set of debugging windows includes tabs "Autos", "Locals", and "Watch" in one panel (usually bottom left corner). If you do not see a window that we discuss here, you can click `Debug -> Windows` and select the window you want to see. Autos display variables used in the previous statement and the current statement. This is useful when you are at a particular line in a program and only care about the subset of all variables when investigating a problem. Locals will list all local variables in the current scope. In the Watch window, you can display *any* valid expression that is recognized by the debugger (for example sum of two variables).

   In **gdb**, you can use `print` to see the values of variables and expressions. Use `display` to specify the values and variables to print every time your program is stopped.

5. **Program flow:** You will find another set of debugging windows in the second panel. These are "Call Stack", "Breakpoints", "Command Window", and "Output". The Call Stack displays the current execution path (in terms of function calls). Click on different entries in the call stack and different code will be displayed. Be sure to get back to the code for `Rectangle`. You can tell from the position of the yellow arrow. Breakpoints lists all breakpoints in your program. Output window shows status of various features in the development environment and we won't use it much in this class. Command Window is very useful and powerful. It allows you to evaluate expressions, execute statements, print variable values, and sometimes even change them. It operates in two modes, we will use only Immediate mode (you should see the tab name as "Command Window - Immediate", if you do not, type `immed`).

   In **gdb**, you can use the command `backtrace` 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 first one (nearest to the top of the stack) that is your code. By typing `frame N`, where `N` is the index of 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.

6. **Command Window** Let's try some of the features of the Command Window. Type `?m_upper_right`. You should see complete report about this variable. Now try to change the +x+ coordinate of the upper right corner by typing `m_upper_right.m_x=2`. Again, check the status of the variable. When you type `?m_points_contained`, you get report about your vector variable. You will see three pointers with their values; these are pointer to an array which is internal representation of the vector. Type `?m_points_contained._Myfirst` and you will find out about the first element (Point2D). To see the second, type `?m_points_contained._Myfirst[1]`, and so forth (until you reach the total vector size).

   In **gdb** many of these options are handled through the `print` and `display` commands.

7. Now, use F10 to step through the execution one line at a line. Look at the source code, the console, and the watch window. You should see the error pretty soon. Use the Watch window to find the bug in the program. Hint: You can even display one coordinate of your rectangle. When you have found it, show a TA how you found it using the debugger. Have your Watch window visible and also show the contents of your Command Window where you tried examples above (enlarge the windows to take about half of your laptop screen). Be ready to answer questions about the purpose of the other debugging windows.

   In **gdb**, you can use the command `display` to see the contents of a particular variable or expression when the program is stopped. Therefore you can complete this part of the exercise easily using gdb.

8. **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.

   Click on a line in the main program that comes after the definition of `pt`, and then go to the top level menu, click on `Debug`, go to `-> New Breakpoint`, and choose `New Data Breakpoint`. In the `Address` field, type `&pt.m_x` (remember `pt` is the name of the `Point2D` object), and ignore the other fields. In this case, the `&` tells the debugger the address of the member variable `pt.m_x` and the debugger will now stop whenever the value at that location is changed. It was very important to click on a line in the function where `pt` is defined; otherwise the debugger will tell you the variable is undefined!

   Finally, click `Ok` and then `Debug -> Continue`. The debugger will now run until the value of `pt.m_x` is changed.

   Show your TA that your program successfully stopped when the variable got changed. Your TA may also ask you questions about the other steps in debugging. This will **complete this checkpoint** and the lab.

   In **gdb**, you can use the command `watch` to halt the program when a variable or expression changes. Therefore, you can complete this part of the lab using gdb. Note that in gdb you do not need to pass a reference for a watch point, so you should just type `watch p.m_x`.

Please note that the 3rd checkpoint has only given you a brief introduction to debugging. You will learn much more through extensive practice. In fact, when you go to see one of us — the TAs or instructor — to ask for help in debugging your assignments, we will constantly be asking you to show us how you have attempted to find the problem on your own. This will include combinations of the four steps listed at the start of the lab.