

# CSCI-1200 Computer Science II — Fall 2006

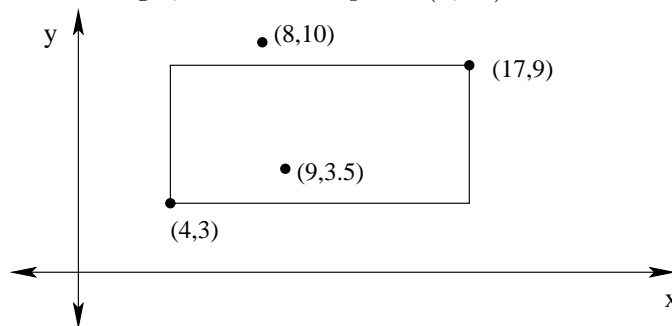
## Lab 4 — Testing and Debugging

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. Judicious use of `cout` statements to see what the program is actually doing (a.k.a. “`printf` debugging”). This is especially useful for printing the contents of a large data structure or class. It is often hard to visualize large objects using the debugger (see next item) alone.
4. 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.

### 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 (a.k.a. *axis-aligned*). This makes it easy to represent and to test if a point is inside. Our code will create points and rectangles and determine which points are in which rectangles. This type of problem is common in graphics and robotics. The rectangle below 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 and then turn off your internet connection:

```
http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/04\_debugging/point2D.h  
http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/04\_debugging/rectangle.h  
http://www.cs.rpi.edu/academics/courses/fall06/cs2/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. Don't worry if you can not, but don't fix it in the code if you do!

**To complete this checkpoint:** Look through `Rectangle.h` and `Rectangle.cpp` to determine what functions need to be added and finish the implementation. Compile these files and remove any compilation errors.

## Checkpoint 2

Create a new file in Visual Studio 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`. When there is non-trivial logic in a function, multiple inputs to the test code should be attempted to test every path through the program. 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 in the provided code. Even if you know how to fix the bugs, please do not fix them yet.

## Checkpoint 3

Now, we need to practice using the debugger to find and fix errors. The instructions below are specific to Visual Studio, but you may use any debugger that that works with your particular development environment. Make sure you learn how to do each of the tasks described below in your debugger. If you are using a debugger other than Visual Studio, you may re-enable your internet connection to read reference material specific to your debugger & development environment.

- 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 will get suspicious about a place in your code that might be wrong.
- **Set a breakpoint:** In the source code, go to the file where error might have originated. Select `Debug -> New Breakpoint`. Four options will be presented. You can set a breakpoint based on the program entering a function, at specific point in a file, at a memory address, or when a specific data condition is met. Click `File` and you will notice that the current line number is displayed. Click `OK`. A breakpoint is now set. Your program will halt when execution reaches this location in the code (when run using the debugger). Note, that you can set this simplest Breakpoint (with standard options) not only using this menu, but also right clicking at a particular line (then `Insert Breakpoint`), or even more quickly clicking on a grey bar along left side of your window.
- 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.
- **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.

- **Content of variables:** The default set of debugging windows includes tabs “Autos”, “Locals”, and “Watch” in one panel (usually bottom left corner). If you don't 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).
- **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`).
- **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).
- 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.
- **Breakpoint on Variable Change:** The last powerful debugger feature you will find out about today is variable monitoring. Create an instance of `Point2D` in your main function and change the coordinates using `Set` function. You will now monitor the change of this point using the debugger. Select `Debug -> New Breakpoint` and choose the `Data` tab. In `Variable` field, type `pt.m_x`, `pt` is my point in this case. `Context` is to tell the debugger, where can the variable be found. The syntax is `{[function],[source],[module]}`, in our case, it is enough if we fill in `{,test_rectangle.cpp,}`. Hit `OK` and run the debugger.

**To complete this checkpoint:** When you have completed the steps above, raise your hand and the TA will quiz you on debugging and ask you to demonstrate one of the tasks.

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 the 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.