Debugging with Xcode

This handout has many authors including Eric Roberts, Julie Zelenski, Stacey Doerr, Justin Manis, Justin Santamaria, and Jason Auerbach.

Because debugging is a difficult but nonetheless critical task, it is important to learn the tricks of the trade. The most important of these tricks is to get the computer to show you what it's doing, which is the key to debugging. The computer, after all, is there in front of you. You can watch it work. You can't ask the computer why it isn't working, but you can have it show you its work as it goes. Modern programming environments usually come equipped with a ***debugger,*** which is a special facility for monitoring a program as it runs. By using the Xcode debugger, for example, you can step through the operation of your program and watch it work. Using the debugger helps you build up a good sense of what your program is doing, and often points the way to the mistake.

This handout is designed for use with Xcode version 4. If you are using an earlier version of Xcode, the screenshots will look a bit different, but the overall strategy will be similar.

Using the Xcode debugger

The Xcode debugger is a complicated environment, but with a little patience, you should be able to learn it to the point where you code more efficiently and productively.

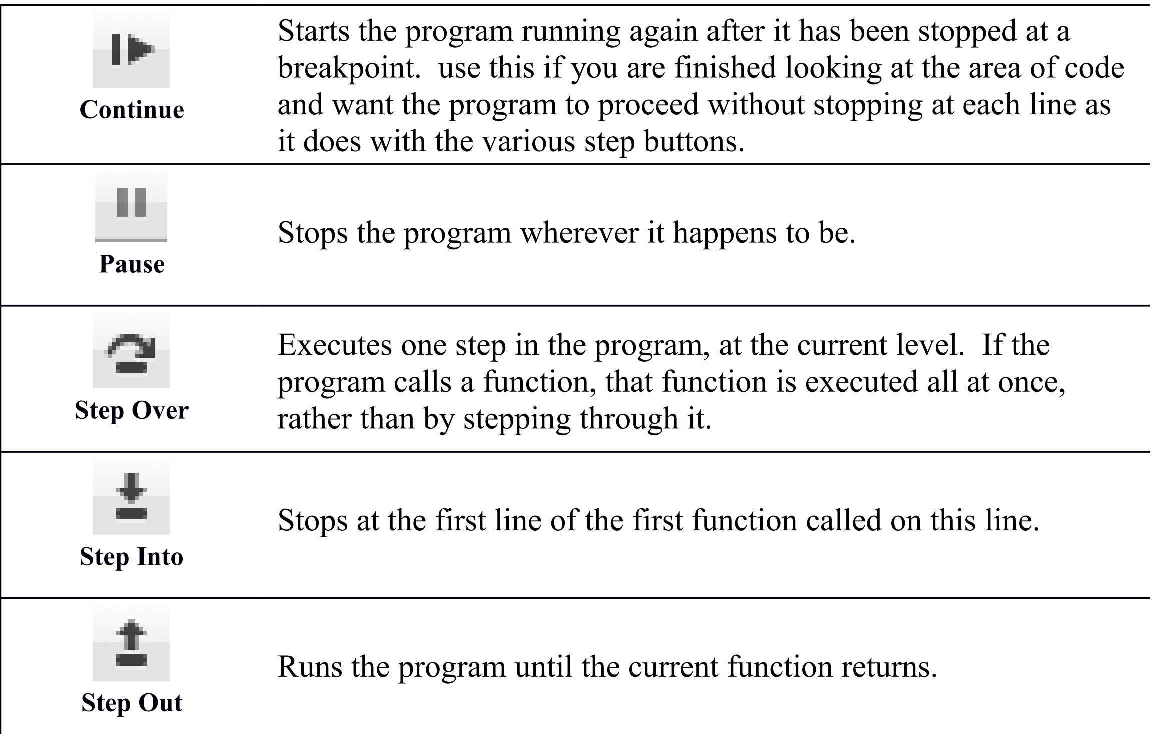
The tool bar for Xcode includes a command **Build and Go** that you can use to debug your program. The mini-debugger gives you the ability to stop your program midstream, poke around and examine the values of variables, and investigate the aftermath after a fatal error to understand what happened. When you choose the **Debug** menu item, it sets up your program and then brings up the debugger window without starting program execution. At this point, you control the execution of the program manually using the buttons on the toolbar in the debugger window. You can choose to step through the code line-by-line, run until you get to certain points, and so on.

When a program starts with debugging enabled, Xcode opens a debugging toolbar containing a set of icons. The toolbar icons you should become familiar with are the first five in the list: **Continue**, **Pause**, **Step Over**, **Step Into**, and **Step Out**. These icons and their corresponding commands are detailed in Figure 1.

The **Continue** will start the program from where it left off. The **Pause** button is useful if the program is in an infinite loop or if you want to stop the program manually to use the debugger. (There is also a **Tasks** button with a stop sign on the Xcode control strip that terminates execution of the program if you want to return to editing.) Ordinarily, the program will continue to run until you click the **Pause** button or until it encounters a breakpoint. Setting a breakpoint (as described below) makes it possible for you to pause the program when it reaches a section of code that you want to investigate in more detail.

Once your program is paused, the bottom pane of the debugger window will show the current function that is executing and a green arrow to the left of the code shows the next line to be executed. Choosing **Continue** causes your program to continue executing. The three **Step** buttons, by contrast, give you more fine-grained control over how the execution proceeds, which makes it possible to watch exactly what's happening as you search for bugs in your code.

Figure 1. Debugger toolbar icons



Icon	Description
Continue	Starts the program running again after it has been stopped at a breakpoint. use this if you are finished looking at the area of code and want the program to proceed without stopping at each line as it does with the various step buttons.
Pause	Stops the program wherever it happens to be.
Step Over	Executes one step in the program, at the current level. If the program calls a function, that function is executed all at once, rather than by stepping through it.
Step Into	Stops at the first line of the first function called on this line.
Step Out	Runs the program until the current function returns.

To get a better sense as to how this entire process works, take a look at the example in Figure 2, which shows a program stopped before the call to the `distanceFromOrigin` function. Clicking **Step Over** executes the next line of code, automatically calling any functions invoked by that line. As a result, you can ignore the details of operations that are at lower levels of the code than the part you are debugging. Clicking **Step Over** at this point would execute the `distanceFromOrigin` call and assign it to `distance` without having to step through the details of `distanceFromOrigin`. The **Step Into** command makes it possible to drop down one level in the stack and trace through the execution of a function or a procedure. In Figure 2, the debugger would create the new stack frame for the `distanceFromOrigin` function and return control back to the debugger at the first statement of `distanceFromOrigin`. The **Step Out** command executes the remainder of the current function and returns control to the debugger once that function returns. **Step Over** executes the next line of code, automatically calling any functions invoked by that line. As a result, you can ignore the details of operations that are at lower levels of the code than the part you are debugging. For example, you would hate to trace through the steps involved in each call to `cout`, and the **Step Over** button allows us to skip all of the details.

```
int main() {
    cout << "This program finds the maximum distance from the origin" << endl;
    cout << "for a set of points. Enter a blank line to stop." << endl;
    double maxDistance = 0;
    while (true) {
        double x, y;
        string line = getLine("Enter x, y: ");
        convertToPoint(line, x, y);
        double distance = distanceFromOrigin(x, y);        Thread 1: breakpoint 1.1
        if (distance > maxDistance) maxDistance = distance;
    }
    cout << "The maximum distance is " << maxDistance << endl;
    return 0;
}

/*
 * Function: distanceFromOrigin
 * Usage: double distance = distanceFromOrigin(x, y);
 * ------------------------------------------------
 * Returns the distance from the origin to the point (x, y).
 */
double distanceFromOrigin(double x, double y) {
    return sqrt(x * x + y * y);
}
```

```
distance = (double) 0
line = (std::__1::string) "3 4"
maxDistance = (double) 0
x = (double) 3
y = (double) 4
```

```
This program finds the maximum distance from the origin
for a set of points. Enter a blank line to stop.
Enter x, y: 3 4
(lldb)
```

Figure 2: Using the Xcode Debugger

Thread switching

Occasionally, when you get an error or press the `Pause` button, the stack trace will be some thing that looks totally unlike any code you've written. This is because the process of running your C++ program actually involves other activities besides the code you have written. Each of these activities is running under the control of something called a ***thread***, which is a particular style of concurrent process. If you are using the graphics library, for example, the program needs a separate thread to make sure that the window is correctly updated. If you find that the stack trace you get after invoking `Pause` is completely mysterious, you may have paused one of the other threads besides your own.

If you get such a stack trace, click on the name of the thread. Doing so will give you a panel showing the currently active threads. Select different threads until you get one that looks like your code.

Using breakpoints

Clicking in the narrow column to the left of the code displayed in the bottom pane sets a ***breakpoint*** at the indicated line. A second or two after you click on the side, a blue pentagon sign appears to indicate that there is a breakpoint set on that line. When the program is started by the **Build and Go** command, it checks to see whether there is a breakpoint on each line that it executes. If there is, the program stops at that point. Clicking on the pentagon removes the breakpoint.

Breakpoints are an essential component of debugging. A general strategy is to set a few breakpoints throughout your program; usually around key spots that you know may be bug-prone, such as computationally intensive or pointer-intensive areas. Then, run your program until you get to a breakpoint. Step over things for a few lines, and look at your variable values. Maybe step out to the

outer context, and take a look to see that the values of your variables are still what they should be. If they're not, then you have just executed over code that contains one or more bugs. If things look well, continue running your program until you hit the next breakpoint. Lather, rinse, repeat.

Getting to the scene of the crime

While running the debugger, it is usually easy to see where exactly a program crashes. On a memory error where a dialog box pops up (such as an "Access fault exception"), the program immediately halts, and the debugger window presents the current state of the program right up to the illegal memory access. Even though the program has terminated, you can see exactly where it stopped, dig around and look at variable values, look at other stack frames and the variable values in those calls, and do some serious detective work to see what went wrong.

A call to **`error`** has a similar behavior. If there are cases which shouldn't happen when the code is running correctly but might if there is a bug, you can add checks for them and call **`error`** if the checks turn out true. This means that if one of those cases occurs, the debugger will stop on the **`error`** line so you can look around and see what's going wrong.

Sometimes it is not obvious at all as to what is going on and you don't know where to start. Errors aren't being triggered, and there aren't memory exceptions being raised, but you know that something's not right. A great deal of time debugging will not be spent fixing crashes so much as trying to determine the source of incorrect behavior.

Imagine you have an array of scores. It is perfectly fine when you created and initialized it, but at some later point, its contents appear to have been modified or corrupted. There are 1000 lines executed between there and here—do you want to step through each line-by-line? Do you have all day to work on this? Probably not! Divide and conquer to the rescue! Set a breakpoint halfway through those 1000 lines in question. When the program breaks at that point, look at the state of your memory to see if everything's sane. If you see a corrupted or incorrect value, you know that there's a problem in code that led to this point. Just restart and set a breakpoint halfway between the beginning of the code path and the first breakpoint. If everything looks okay to this point, repeat the process for the second half of the code path. Continue until you've narrowed the bug down to a few lines of code. If you don't see it right away, take a deep breath, take a break, and come back and see if it pops out at you.

Building test cases

Once your program appears to be working fine, it's time to really turn up the heat and become vicious with your testing, so you can smoke out any remaining bugs. You should be hostile to your program, trying to find ways to break it. This means doing such things as entering values a user might not normally enter. Build tests to check the edge-cases of your program.

For example, assume you've written a function

```
generateHistogram(Vector<int> & buckets, Vector<int> & scores)
```

where the **`buckets`** vector represents uniformly sized ranges that together cover the range from the minimum score to the maximum score, which are given by constants. When a score falls in the range of that specific bucket, the bucket is incremented by one.

An example of an edge case to test would be to have 0 scores. Does the function handle the case where one or more of the score values may be zero or negative? More than 100? What if the difference between the lowest score and highest score is not evenly divisible by the number of buckets? Thinking

of the assumptions you've made about the input to a function and writing tests that violate those assumptions can lead to a healthy testing of edge cases.

Seeing the process through

One of the most common failures in the debugging process is inadequate testing. Even after a lot of careful debugging cycles, you could run your program for some time before you discovered anything amiss.

There is no strategy that can guarantee that your program is ever bug free. Testing helps, but it is important to keep in mind the caution from Edsger Dijkstra that "testing can reveal the presence of errors, but never their absence." By being as careful as you can when you design, write, test, and debug your programs, you will reduce the number of bugs, but you will be unlikely to eliminate them entirely.