CS106B Handout #7M
Winter 07-08 January 14, 2008

Debugging with Xcode

This handout was originally by Eric Roberts and Stacey Dorr. Revised by the Justins and Julie, and again by Jason and Julie

> *As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*
>
> —Maurice Wilkes, 1949

The purpose of this handout is twofold: to give you a sense of the philosophy of debugging and to teach you how to use some of the practical tools that make testing and debugging easier.

The philosophy of debugging

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. To a large extent, the problems that people face debugging programs are not so much technical as they are psychological. To become successful debuggers, you must learn to think in a different way. There is no cookbook approach to debugging, although Nick Parlante's rules below will probably help. What you need is insight, creativity, logic, and determination.

The programming process leads you through a series of tasks and roles:

Design	—	Architect
Coding	—	Engineer
Testing	—	Vandal
Debugging	—	Detective

These roles require you to adopt distinct strategies and goals, and it is often difficult to shift your perspective from one to another. Although debugging is extremely difficult, it can be done. It will at times take all of the skill and creativity at your disposal, but you can succeed if you are methodical and don't give up on the task.

Debugging is an important skill that you will use every day if you continue in Computer Science or any related field. Even though it is the final task of those listed above, it is certainly not the least important. Debugging will almost always take more time than the first three tasks combined. Therefore, you should always plan ahead and allow sufficient time for testing and debugging, as it is required if you expect to produce quality software. In addition, you should make a concentrated effort to develop these skills now, as they will be even more important as programs become more complicated later in the quarter and later in your career as a programmer.

The Eleven Truths of Debugging

1. Intuition and hunches are great—you just have to test them out. When a hunch and a fact collide, the fact wins.

2. Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable producing extremely simple and obvious errors from time to time. Look at code critically—don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.

3. The clue to what is wrong in your code is in the values of your variables and the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.

4. Be systematic. Be persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.

5. If you code was working a minute ago, but now it doesn't—what was the last thing you changed? This incredibly reliable rule of thumb is the reason your section leader told you to test your code as you go rather than all at once.

6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable at a time. It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs.

7. If you find some buggy code which does not seem to be related to the bug you were tracking, fix the bug code anyway. Many times the buggy code is related to or obscures the original bug in a way you had not imagined.

8. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions which led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your functions cannot contain the bug. One of these arguments will contain a flaw since one of your functions does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.

9. Be critical of your beliefs about your code. It's almost impossible to see a bug in a function when your instinct is that the function is innocent. In that case, only when the facts have proven without question that the function is the source of the problem will you be able to see the bug.

10. You need to be systematic, but there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the functions you suspect the most first. Good instincts will come with experience.

11. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. Many times a programmer can spend

> hours late at night hunting for a bug only to finally give up at 4:00A.M. The next day, they find the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the "go do something else for a while, come back, and find the bug immediately" scenario happens too often to be an accident.
>
> — Nick Parlante, Stanford University

Testing incrementally

When dealing with any program containing more than a couple of functions, it is highly recommended that testing be done on an incremental basis. This might include writing test code that passes values to a function, and gets its return value. Doing this allows you to test your functions in isolation, allowing you to hone in on bugs before putting everything together. Realizing there's a bug in a specific 10-line function by seeing the results of a very simple test program is much easier than testing it in the context of the rest of your program, where it is no longer isolated and much harder to tell whether a bug is a result of that function, or any other one. This makes debugging much less painful in the long run.

Keeping an open mind

For programs that use random numbers, it is important to debug the program in a deterministic environment. A deterministic environment is one that allows a piece of code to behave exactly the same way every time you run it. For this reason, it is essential to take the `Randomize` statement out of the main program, usually by enclosing it in comment markers. You want it there eventually, but not while you're debugging. You want your program to work the same way each time, so that you can always get back to the same situation.

Avoiding value rigidity

One of the most important techniques to master about debugging is keeping an open mind. So often, the problems that keep your code from working are easy to see if you can simply overcome the psychological blinders that keep you from seeing them. The more you are sure that some piece of code is correct, the harder it is to find the bugs in it.

Dealing with pointers

Programs involving pointers can be very tricky to debug, because errors in pointer handling can lead to very strange consequences. With pointers, you can delete something twice and/or use a variable after it has been freed, run off the end of the array, etc., and you can get away with it at the time, but later the ill effects begin to show. This is why it is so important to have your memory model straight before you start coding, and it doesn't hurt to make a memory diagram to make sure you understand what you want, and compare it to what you have actually coded. Also, be extra careful when scrutinizing allocations and initializations of pointers.

Debugging with `cout`

Interspersing code with `cout` statements can be a quick and easy way to check the value of a variable name at a certain point in a program, or to see whether or not a specific point of execution is being reached. It is important to note that the way `cout` works may not be quite what you would expect. Usually there is a buffer that stores the data piped to calls to **`cout`**. The data in the buffer is only displayed on the screen once **`endl`** is sent, or a function requiring input is called (like **`GetInteger()`** or **`GetLine()`**).

For example:

```
int main() {
      int i, int j;

      i = ComplicatedFn();
      cout << "got past ComplicatedFn(); i = " << i << ".";        // debug

      j = ComplicatedFn2();
      cout << "got past ComplicatedFn2(); j = " << j << ".";       // debug

      ComplicatedFn3(&i, &j);
      cout << "done: i = " << i << ", j = " << j << endl;          // debug

      return 0;
}
```

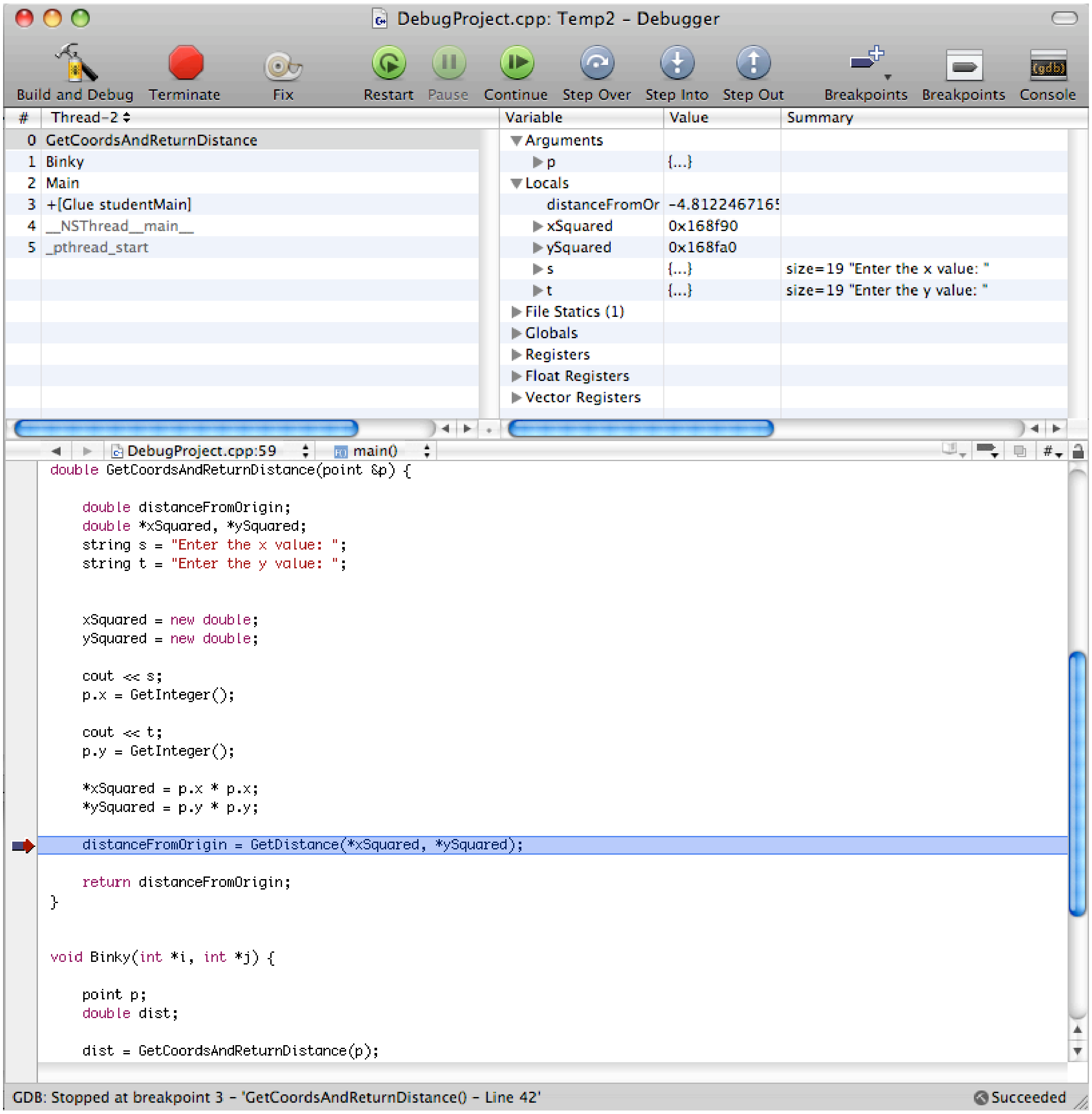
Assuming no other i/o calls are in the other functions, no **`cout`** statements will print until the last **`cout`** call is executed, since it has the first (and only) **`endl`** call. Therefore, if your program crashes in the middle of **`ComplicatedFn2()`**, the output window will not have printed out anything, even though a **`cout`** call had been passed. Therefore, it is always a good idea to attach **`endl`** to all debugging statements, so the buffer is flushed and the debug statements print to the screen when they are passed.

Even though **`cout`** statements may seem like a sufficient means for debugging, it becomes largely inadequate for more complicated programs, especially those which make many iterations over the same code during a single program run. In cases where **`cout`**-style debugging is largely inefficient and/or inadequate, we turn to the debugger.

Using an online debugger

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.

Using the Xcode debugger



#	Thread-2
0	GetCoordsAndReturnDistance
1	Binky
2	Main
3	+[Glue studentMain]
4	__NSThread__main__
5	_pthread_start

Variable	Value	Summary
▼Arguments		
▶p	{...}	
▼Locals		
distanceFromOr	-4.812246716!	
▶xSquared	0x168f90	
▶ySquared	0x168fa0	
▶s	{...}	size=19 "Enter the x value: "
▶t	{...}	size=19 "Enter the y value: "
▶File Statics (1)		
▶Globals		
▶Registers		
▶Float Registers		
▶Vector Registers		

```
double GetCoordsAndReturnDistance(point &p) {

    double distanceFromOrigin;
    double *xSquared, *ySquared;
    string s = "Enter the x value: ";
    string t = "Enter the y value: ";


    xSquared = new double;
    ySquared = new double;

    cout << s;
    p.x = GetInteger();

    cout << t;
    p.y = GetInteger();

    *xSquared = p.x * p.x;
    *ySquared = p.y * p.y;

    distanceFromOrigin = GetDistance(*xSquared, *ySquared);

    return distanceFromOrigin;
}


void Binky(int *i, int *j) {

    point p;
    double dist;

    dist = GetCoordsAndReturnDistance(p);
```

GDB: Stopped at breakpoint 3 - 'GetCoordsAndReturnDistance() - Line 42'

Succeeded

Figure 1: The Xcode Debugger Window

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.

Under the Build menu, there are two menu items for executing your code: the Run (**Cmd-R**) option and the Debug (**Cmd-Y**) option. So far you've probably been using Run, which sets up and runs your program on its own until it completes successfully or halts abnormally because of a fatal error. The Debug option, on the other hand, will set up and run your program in the context of the debugger. The debugger gives you the ability to stop your program mid-stream, 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 debugger window. You can choose to step through the code line-by-line, run until you get to certain points, and so on.

The toolbar icons you should become familiar with are Build and Debug, Terminate, Restart, Pause, Continue, Step Over, Step Into, and Step Out.

Build and Debug	Build and Debug	Runs the program, which continues until you choose Stop or the program encounters a breakpoint, discussed below. The shortcut key for this is **Cmd-Y**.
Terminate	Terminate	Stops the program and debugger immediately.
Restart	Restart	Starts the program over. This is most useful if you have made some changes to the code and want to see how they affect the program's execution.
Pause	Pause	Stops the program wherever it happens to be.
Continue	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 "step" buttons.
Step Over	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	Step Into	Stops at the first line of the first function called on this line.
Step Out	Step Out	Runs the program until the current function returns.

Whenever the program has not yet been started, the Build and Debug command will start it. If it has started but has stopped, Continue will start it again 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. The Terminate command is useful if you want to want to quit the debugger and return to editing before the program has finished running.

When a program starts with debugging enabled, the debugger window opens. When it does so, the program is either running or preparing to run. If you quickly click on the Terminate button, you can stop the program through mid-execution and see what is happening right at that point. Setting a breakpoint (discussed below) will automatically stop the debugger at a specific line.

Once your program is stopped, the bottom pane of the debugger window will show the current function that is executing and a red arrow to the left of the code shows the next line to be executed. Choosing `Continue` would cause your program to continue executing freely while the three Step buttons allow you to manually control what to execute and then return control to the debugger. `Step Over` is useful whenever you know that the function on that line works and that the bug is further on. 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. In Figure 1, calling `Step Over` would execute the `GetDistance()` call and assign it to `distanceFromOrigin` before control returns to the debugger after executing that line. 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 1, the debugger would create the new stack frame for the `GetDistance()` function and return control back to the debugger at the first statement of `GetDistance()`. The `Step Out` command executes the remainder of the current function and returns control to the debugger once that function returns.

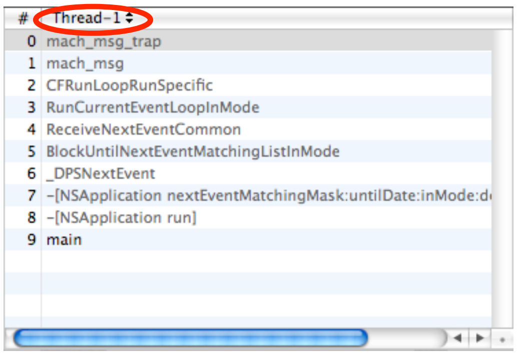
At this stopped point, there are three panes in the window filled with information. The left pane shows the current stack frame. Here we see what calls have been made that have led to the current call. The most recent call is listed at the top, while `_pthread_start` (which is an Xcode system function) is at the top. In Figure 1, we see that `main()` called function `Binky()`, which called `GetCoordsAndReturnDistance()`. When we click on a function listed in this pane, the frames on the right and bottom (discussed below) will update to show the values and information for the selected function. This essentially allows you to look at the entire state of your program, not just the current function, any time the program is stopped.

The right pane contains all variables and associated values contained in the stack frame highlighted in the left pane. In Figure 1, we see `distanceFromOrigin` has quite a strange value. Looking at the code (and using our mad debugging skills) we see that `distanceFromOrigin` has yet to be initialized at this point in execution and thus has a junk value. Also note that some variable types (such as structs, classes, and pointers) have an arrow to the left of them. This means that there is more information pertaining to this data type that can be viewed. For instance, by clicking on the arrow next to `p`, which is a variable of type `point`, we see the values for the x and y variables, which are members of the `point` struct. The value of `ySquared` is `0x168fa0`. This is the address (in hexadecimal) of the chunk of heap memory to which it is pointing. This may be interesting, but usually not what we want. Clicking on the arrow to the left reveals what is truly useful, the value of `*ySquared`, set to `16` in this case. In Xcode, when a variable changes values it is highlighted in red, so as to inform you of the update. A highlighted value returns to black once the next call is executed.

The bottom pane contains the program's code with breakpoint information and current point of execution. The red arrow in the bottom frame indicates the next line of code to be executed. Clicking on functions listed in the stack frame will adjust this window accordingly and change the red arrow to indicate the next statement to be executed for that particular function.

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 compiler is actually running a few other "threads" of code other than your own to enable the code to run properly. If you get such a stack trace, click on the name of the thread:



#	Thread-1 ⇕
0	mach_msg_trap
1	mach_msg
2	CFRunLoopRunSpecific
3	RunCurrentEventLoopInMode
4	ReceiveNextEventCommon
5	BlockUntilNextEventMatchingListInMode
6	_DPSNextEvent
7	-[NSApplication nextEventMatchingMask:untilDate:inMode:d
8	-[NSApplication run]
9	main

You will get a list of different threads. Select different ones until you get one that looks like your code (usually it will be Thread-2).

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 Debug** 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. Wash, 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. **`Error()`**s 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/incorrect value, you know that there's a problem in code that led to this point – restart and set a breakpoint halfway between the beginning of the code path and the first breakpoint. If everything looks ok 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 **`Histogram(Vector<int> &values, Vector<int> scores)`** where each bucket represents a range from the minimum score to the maximum score. 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. Another would be to have 0 buckets. 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 an odd number? 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.