This handout originally written by Nick Parlante.

The 10 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. The values of the variables are never wrong.

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. Try to see what the facts are pointing to. Be systematic and persistent. The bug is not moving around in your code, trying to trick or evade you. It is just siting in one place, doing the wrong thing in the same way every time.

4) If you code was working a minute ago, but now it doesn't— what was the last thing you changed? Note: 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. If you run your program each time you've added 50 more lines of code, then you'll almost always know which 50 lines are responsible as soon as things begin to not work.

5) 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 much more difficult to interpret, and you tend to introduce new bugs.

6) If you find some wrong code which does not seem to be related to the bug you were tracking, fix the wrong code you found anyway. Many times the wrong code was related to or obscuring the bug in a way you had not imagined.

7) 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.

8) Be critical of your beliefs about your code. It's almost impossible to see a bug in a function when your instinct is that it is innocent. In that case, only when the facts have undeniably proven the function is the source of the problem will you be able to see the bug.

9) 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.

10) Debugging requires an objective and reasoned approach—a global perspective and clear understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you unsuccesfully pursue a bug, the less perspective you tend to have. Realize when you have lost the ability to debug. Take a break. Get some sleep. It often happens that after spending late hours hunting for a bug only to give up at 4:00 a.m, you find the bug the next day in just 10 minutes. What allowed you to find the bug so quickly? Maybe you just needed some sleep and fresh perspective. Maybe your subconscious figured it out while you're asleep. In any case, "do something else, come back, and immediately find the bug" works too often to be chance.

Show me where your code hurts

When you write a program, you have a mental plan in mind of what you are trying to compute. The program itself is an extremely unambiguous plan of some computation. A bug is when these two plans diverge. Unfortunately, you do not observe the bug directly. Instead you see a symptom produced by the underlying bug. Your job is to work backward from the symptom to find the bug which produced it. The line in your code which contains the bug will have the following property. Before that line, the values of your variables were ok. After that line executes, some of the values are wrong.

There are three basic themes in trying track down the bug. You can work backward linearly from the symptom trying to find the line which took in good values but produced bad values. You can trace forward from the beginning of your program, looking for the values to go bad. Or you can do a sort of "binary search" in your code with break points, narrowing in on the bug by seeing if it lays before or after each trial breakpoint. People develop a personal craft for how they like to debug— the key is to be methodical.

What you want to avoid is the worst case: you've probed around in your code for a few hours without success. You've looked at a lot of your functions but can find nothing wrong. You begin to change around the code in functions in an effort to fix the elusive bug. The problem here is that you are not really getting any closer to finding the bug. It's sort of like looking for a diamond ring on a football field. If you just stroll around at random you may find it. But if after a couple hours you haven't found it-- then are not much better off than when you started since you don't know where you've looked and where you haven't. The most important quality about the following strategy is that by being systematic, you always know which part of the field you've checked.

The Solution

Identify the line which corresponds to the symptom. This is typically is very simple. For the above example the symptom corresponds to the line where the variable went out of bounds or to the printfs which produced the garbage output. The incorrect behavior will correspond to a variable with a bad value. Identify the variable(s) with bad values. For this example, suppose that the program has a problem on line 112 because i is -1 and that is out of bounds for an array.

The critical question is: where did i get its value? There are basically three ways a variable can get a value: the variable appears on the left hand side of a =, a reference to the variable is passed to somebody who changes it, or there is a bad pointer reference somewhere which is accidentally scribbling on the variable. Arguably a fourth way is if the variable is never initialized and so gets a random value. Writing your code with good style: sensibly used parameters, no global variables should make it easier to identify what bit of code could be changing your variable.

In this example, i is wrong on line 112. Look above line 112 searching for the most recent line which changes i. Suppose line 108 passes i as a reference and so is suspect. Put a breakpoint or printf just before line 108 to examine i. If i is correct before 108 but wrong after, then 108 is the problem. If i was already wrong, then you must probe further backward. If working backwards doesn't reveal the good values/bad values transition, it can be easier to sprinkle prints or asserts at the very beginning of a few functions just to find the general area where things are going wrong.

One starting strategy which doesn't require any breakpoints is to step over the functions in main() without stepping into any one. Keep an eye on your data state after each step to see which function messes things up. Of course, if you didn't decompose out the basic phases of your program, then this won't work.

Deer in the Headlights

The key is not to lock up in deer-in-the-headlights mode when overwhelmed with the enormity of the problem. You are never more than a couple well-placed break point examinations away from seeing the problem. Try thinking about which breakpoint will implicate/exonerate the largest amount of code with the least work.