15-122: Principles of Imperative Computation

Recitation 7 — Josh Zimmerman

Unit testing

(Note: A large amount of content in this handout comes from Jonathan Clark, who TAd 15-122 in Fall 2012.)

It's really important to always test your code as you write it. Why, you ask? Well, it's effectively impossible to write bug-free code all of the time. However, you want your code to work. So, testing your code is essential to help you find and eliminate bugs.

Why should you do it as you write it? The sooner you catch a bug, the easier it will be to fix. If you wait a long time between writing code and testing it, you might not remember exactly how your code is supposed to work. Not understanding exactly how your code is supposed to work makes debugging much harder.

In addition, if you write a buggy function `foo` and then write a function `bar` that calls `foo`, you might see incorrect behavior in `bar` that is actually caused by the bug in `foo`. This kind of situation would make tracking down the actual source of the bug far more difficult than it has to be.

So, unit test your program! If you test just one function at a time, it'll be far easier to determine where the problem in your program lies.

Here are some major important points about unit testing.

- Use unit tests to look at edge cases!
  - Edge cases are inputs that the specification for your function allows but that might be tricky to handle. They're a common source of bugs because you may need to handle edge case inputs differently from normal cases.
  - Some common edge cases with `int`s are `int_min()`, `int_max()`, 0, -1, and 1. (These are not all of the cases! Depending on what function you are writing there may be other inputs that are edge cases.)
  - For arrays, some edge cases you can run into are the empty array, an array of length 1, and very long arrays. Again, these aren't all of the edge cases — what the edge cases are depends on what the function you wrote is.
- Use unit tests to help you narrow down where exactly bugs in your program lie.
  - If all that you know is that your program produces the incorrect result, it's essentially impossible to debug it. If you write good unit tests for all of your functions, you'll be able to run those and narrow down the bug to specific function/functions, thus making your code far easier to debug.
  - Related to this, you should write tests that exercise all of your code: If you have an `if` statement with two branches, make sure you have a test for each branch. If you don't, one of them might be incorrect and you'd never know it. (Or you'd have a hard time tracking it down if you think whatever bug there is comes from somewhere else.)
- Use unit tests to make sure your contracts pass when they should pass and fail when they should fail

- Contracts are very useful, but only if they're correct. Incorrect contracts could lead you to thinking that part of your code is wrong when it actually is right, or vice versa. If you test your contracts, you'll be able to make sure that they are correct and won't mislead you.

Taking time to write good tests can save massive amounts of debugging time, since good tests will tell you exactly where in your code the bug lies.

When testing your code, you should start with the assumption that it's incorrect for some input, and that you want to find that input. It's critical to think about corner cases and edge cases — those are things that are out of the ordinary, but are allowed by the preconditions of your function.

If you're trying to test code, it's really important to come at it with the attitude that you want to break it. Pretend your worst enemy wrote the code, and that you want to show them all of the problems with it to get them back for that thing they did, even the pedantic weird cases that probably won't come up in practice. (This is necessary because in the real world you will have to deal with edge cases: if you don't, your code will break for at least some people [but probably thousands], possibly making whatever you wrote unusable for them.)

Now, let's get some practice with testing.

Practice!

Let's work together! I have a few functions, with known specs. Let's try to find all of the cases that they're broken in without looking at their source code.

Then, if there's time, we can use our knowledge of the cases that they're broken in to fix them.