Guide to Coding Iteratively (CS 250 focused)

So, you need to create a program that’s a bit larger than an APT-style problem for which you have little to no starter code for. What do you do? Sure, you could spaghetti code it—i.e., write all the code at once, quickly, and then spend somewhere between a few minutes and eternity debugging. But what if I told you there’s a better way?

Now, better is of course a subjective term, but, generally, an incremental (as opposed to spaghetti) style can promise you a shorter debugging time, easier debugging experience, and cleaner resulting code. But, to each their own.

Now, how do I code incrementally? Well, think back to your first Computer Science lesson. I’ll hazard a guess that it had something to do with printing the words “Hello World” in one of a multitude of languages. That’s what we want to start with.

But, before you even write a print statement, I recommend beginning your coding process on paper. Make sure you understand what program you are trying to implement. Walk through the test cases (if provided) or ideate potential use cases. What goes in and what comes out?

Once you understand what you need to make, you can start *thinking* about coding it. Either in words or psuedocode, plan out the program at a high level. What functions (or subcircuits) do you need? What do they take in and what do they spit out? What built-in functions from your language of choice will you need? Do you know how to use these?

Once this is all sketched out, it’s almost time to begin coding. Now that you know what you need to code, you can break it into bite-sized pieces. Ideally, each task will be indivudually testable and build upon your prior work (e.g., you have to construct a linked list before you can sort it).

First, Hello World your project. Create a main function (or circuit) and ensure it can run without error. Perhaps you can print “Hello World.” After this, your tasks can be focused on the problem at hand. The way you break these next tasks up will be very dependent on the overall problem, but I can outline some general strategies.

* Are there any very basic test cases you can think of? Make sure your program can handle the smallest/most basic sorts of inputs. If it takes in a number, make sure it works for 0 and 1. If it takes in more advanced data, try for one input.
* If it takes in data, can you read one, store it, and print it out successfully? You can’t do much of anything with bad/no inputs.
* If you need to store multiple pieces of data, what is your data structure? Can you add multiple pieces of data to it and then successfully print out the lot?
* Once you have the inputs/whatever data you need to “do something” to, it’s time to do the things. If there’s some sort of mathematical or other manipulation, now would be the time. See if your transformations are correct and can be proved by printing out your data after.
* Do you need to implement some sort of algorithm? Write it out on paper and run through a couple of test cases before implementing it.
* If there are multiple layers to your program (e.g., multiple instructions/types of tasks it must complete), start with the very simplest one, make sure it works, and work your way through them.

My biggest piece of advice is to TEST, TEST, TEST. After every change you make, make sure your program has no id’ed errors, compiles, and runs. It’s a lot easier to get help for one tiny error when you know everything else works than to wait til all the code is written and then solve all the bugs at once. Coding iteratively feels like a larger investment of time up front, but, in my experience, leads to a shorter overall programming experience. If for no other reason, keep these strategies in mind if you become a software engineer—your coworkers are not going to want to approve a 400+ line merge request.

Ex. One tried and tested method to tackle the MIPS homework (there are many ways to do this homework iteratively)

1. Reread your C code, making sure you understand it at a high level and understand the purpose of each line
2. Make a note of all the helper methods you are going to need
3. Choose a simple helper method to start with (like strcpy or strcmp) and write it
4. Push to git
5. Test your simple helper methods on their own, making sure that they work for edge cases
6. Make sure you can read in your inputs correctly and output them again to make sure they are in the correct format
7. Push to git
8. Choose a data structure and figure out how to put your data in the structure
9. Print your structures out as you are creating them, making sure that the data is correctly input
10. Push to git
11. Tie together your structures in a linked fashion, without sorting
12. Push to git
13. Try printing your linked data structure from beginning to end, and make sure that no data is lost in the linking process
14. Push to git
15. Tackle sorting, one case at a time, testing after implementing each case (if you mess up your code terribly, go back to a previous git version)
16. Push to git