Contents

Version 0.1

Lab 2: How do I write this thing?

By now, you may be fighting a feeling of panic or impending doom. There is a way to do this lab in a time efficient manner that many students have used to get good results. A few guidelines may be in order:

- Design first, then code
- Don't try to do the whole thing at once
- Build a little, test a little
- Start early, work steadily (leave it and come back many times)

By Inclusion

Int information in "Doing_lasbs_in_this_class.pdf" and "Thoughts_on_Lab_2_and_Single_Purpose" are officially part of this lab as well as lab 3 and lab 4. *Carefully* read the first of those and go over it.

Design

If you can't design it, you can't code it. Go over the write-up and start writing down questions. Go over it again and see if you can answer any of them and take the rest to office hours. Design, like code, follows the second bullet point above – don't do the whole thing at once. When designing at a higher level, presume the lower level stuff will do its job. You might want to start with the low-level stuff first and build up a set of "bricks" to make the final "wall." Most of us can't pick up a brick wall but nearly all of us can pick up one brick.

An example of low level design: Can you write a function that takes an array of 4 doubles and prints them all on one line? This is the core brick for text output and once you've looked at printf, you can probably design and code this quite quickly. Once this handy brick is available, you can cut loose on the physics loop because you can print the ball and see how the equations change it. You don't even have to get the formatting perfect the first time out of the box – that can be fixed later, but getting numbers out is critical to knowing what your software is doing.

Doing a small chunk, like creating an output routine, and then testing it supports the "build a little, test a little" paradigm. That particular brick of software further supports other build a little, test a little, efforts because it lets you see what happens to the numbers as you add to the physics code.

Kill Syntax Errors Early and Often

When writing C code for this lab, consider stopping after your write each function, saving the file, and compiling just that file. This is to get rid of syntax errors early and to get rid of them before they build up to a soul-crushing heap of errors. Consider this command: **make -r lab2.o** and think about what will happen. Kill syntax errors early, kill them often. The above command doesn't try to make the whole lab, it tries to make one object file. If you use multiple windows, this is easy. In the editor window, save the file but leave the editor live. For vi users that's **:w** and not **:wq**. In your other window, run make to build the .o file that corresponds to the C code file you are editing. Look at the syntax errors in one window and go fix them in the other window. We do it this way because right after you write a function, you are thinking about it. So compile it while it is already in your head. Compiling later means you have to go and figure out what it does all over again, wasting time. 4 windows open at a time is a really good idea in this class.

Potential Plan of Attack

Every one of the tasks below could be a prototype that you keep around. You need 4 working ones to turn in and you will benefit from having more.

1. Create a function that gets passed and array of doubles and prints them on one line.
2. Create a function that declares and initializes an array of doubles and calls the above function to print them.
3. Create a function that takes an array of doubles and runs the equation of motion for X on the X value. Test it with an extended version of item #2 – add a call to this function and a second call to output the array after that.
4. One at a time add the other three equations to the above function, testing as you go.
5. Write a function that returns true if the Y value is zero or more. Test it with two different hard-coded arrays.
6. Take what you have at step 4 and make a loop to repeatedly update the values and print the values. The loop terminates when the function in item 5 returns false. Print the values after the loop ends to show the ball is off the table. This gives parabolic motion without walls to hit.
7. Write a function that uses scanf to read in an unsigned char and the 4 doubles and use the output function to test it. Ignore the unsigned char for now.
8. Use the above to have code that reads one ball and then run the motion code loop on it.
9. Change the scanf code to loop on the scanf call as long as scanf returns 5.
10. Add a function call in the motion control loop right after the call that updates the 4 values. This new function one at a time make calls to the three constraint functions you will need. Do those one at a time. (The constraint functions can tell if the ball hit a particular wall and fixes the position. Your constraint functions will have print statements if they fix the ball position so you can see when they act.)

There's more to the lab than this, but this sequence never add very much code at any given step. At this point if you caught COVID and turned in this much, you'd have a salvageable score (assuming that your code follows the 10-line limit, one-purpose comments, and single purpose functions guidelines).

At this point, you have the lowest level code or something very close to it. You need to enhance your code so that things like the bitfield get put to use. You'd need to add a clock. Re-read the writeup to work out what the top-level stuff looks like and do that. Add graphic output last.