Doing Labs in C in CSE 2421

This document applies to labs 2-4 in Professor Kirby's class. The labs are teaching more than how to program in C. They are also teaching you how to create large programs effectively. The material here is your survival guide for lab 3 especially, but it also applies to labs 2 and 4. It makes lab 2 easier to do but it also makes lab 3 even possible to do. It can save countless hours of frustration in lab 4.

"One Job!" – Single Purpose and Function Length

Each function that you write should have a single purpose. Before you write any function, you should have a crystal clear idea of what that purpose is. You may have heard the phrase, "You had one job…!" and that should apply to every function that you write.

- Functions that go above 10 lines* of code are too long unless approved by a grader or the instructor. Points will be deducted.
- Above each function goes a comment stating the one job the function does or points will be deducted.

The one job comment should be written before you write the function! Graders and the instructor have the right to tell you to go away and put them in and come back after you have done so. Office hours are precious and many people have need of them. There is a correlation between people who don't write the one job comments and people who don't know what they really expect their code to be doing. Some functions have clever names that tell you exactly what they do; they still owe a nice comment. Most commonly the comment is a single sentence. If you get past three sentences, you may have detail control problems. These are easier to fix while you are writing comments than they are while writing code.

You can design by comment! The way this works is to write the one job comments for every function that you need. Under each one, list any function names that the current function will hand work off to. Have each function do a single thing. Do not write any C code, writing C is tedious and painstaking. The goal is to design cheaply in a fluid media that carries enough information that you could turn it into code later. This gives you a roadmap. You can ask if you have missed anything, you can re-organize, and you don't have to compile anything.

To make this all happen, you will need to write numerous functions. Each should "peel one layer off the onion." *Short, simple functions are both easy to write and easy to prove right.* Each function should own only the details it needs to do its job and hand off everything else to helper functions that all know how to do their part.

This will lead to multiple levels of calls and that is OK. At every function along the way the level of abstraction and level of detail change. At a high level, the job of a function might be to run a simulation one time. At a very low level, a function might use the basic equations of linear motion to compute a new position in one dimension. The high level function might have a single variable, a simulation

structure. The low level function deals in position, velocity, and time. Note that the high level function does not know if it is in a loop if an even higher level function is running multiple simulations in succession. Coding like this gives you separation of concerns.

A skill being taught here is the ability to change the level of abstraction and detail as you go. Consider making a phone call as an example. At the highest level, we break the task down:

- Find phone
- Turn on phone
- Call somebody

At this level our "data" is whatever phone we find and the id of whoever we want to call. At this level, the code doesn't know or care if it is the highest level task or not. It doesn't know if it is being called repeatedly by some higher level. At this level, the code doesn't care if the phone is Android or Apple or POTS (Plain Old Telephone Service, a/k/a "landline"). The code at this level doesn't care if the phone is in service or what rate plan is being used. Lower level code will figure this out and deal with it.

Your code should be quick to say, "Not my problem. I have minions for that." Likewise, your code should also be quick to push back and say, "Above my pay grade."

The code that dials the phone doesn't care how you got this particular phone; it might have been in your pocket, it might be borrowed from a friend, and it might have been sitting unused at a desk with no one around. That was somebody else's problem. The dialer really doesn't want the details of how you came up with this particular phone because all it worries about is making sure that numbers go out to the network. Chances are very good that the dialer won't actually dial; it will have the correct dialer minion do the actual dialing depending on what kind of phone it is.

What we want are short, simple functions that oversee a single concept.

*Ten lines. The ten lines are functional lines. Declarations and comments do not count towards the ten line limit. DEBUG prints [see below] do not count towards the ten line limit. An if statement counts but the else usually does not. An else if counts because it's also an if statement. Opening and closing { } do not count. This limit is an adaptation of NASA/JPL coding guideline that says functions must fit on one page. If you can't fit a function on one screen you have a problem. To make it enforceable, we have the ten line limit.

https://medium.com/better-programming/the-power-of-10-nasas-rules-for-coding-43ae1764f73d

We will use things like treat all warning as errors – this is where the zero points rule comes from for labs that do not compile cleanly.

We won't be using all of the NASA/JPL guidelines. In fact, we will actually violate some of them on purpose. Our code isn't going to Mars, but one day you might be writing code that does.

https://www.theengineer.co.uk/software-error-may-have-doomed-esa-mars-lander/

https://www.bugsnag.com/blog/bug-day-mars-climate-orbiter

In spaceflight, software bugs can lead to RUD (Rapid Unanticipated Disassembly) events. In this class, overly complex and lengthy code leads to code that can't be debugged.

Prototypes

The 3 C code labs require prototyping. You will write at least 4 prototypes for each of labs 2-4 and turn them in. The goal of a prototype is to prove that you can do something that you are not sure that you can do. This controls risk. Controlling software risk is a valuable professional skill. Agile programming has this as a high level goal. Chances are that you have never written code to read in an integer or a double precision floating point number in C before. The smart student writes a prototype to prove in code that reads these numbers.

A prototype is a short chunk of code that is allowed to ignore various coding standards so that it can get one job done. Think of it as a test stand and a device under test. The test stand uses hard-coded data and other ugly shortcuts to feed the device under test what it needs. The test stand is strewn with debugging statements that amount to, "here is the world before I do this thing. I am about to do this thing. I just did the thing. Here is the visibly changed state of the world after doing the thing." It is likely that the device under test will have a few debugging statements as well.

What you pick to prototype is up to you. You are required to write 4 of them for each lab, so feel free to prototype anything that makes you feel unsure. It is perfectly fine to have more than 4! You have to have 4 that work and are present and marked as to be graded in your makefile so that graders know what to look at. It's OK to have more than 4 marked; once the graders hit 4 that work, they go on.

Use lab 2 as prototyping practice because labs 3 and especially 4 are far easier with effective prototyping. Use your prototyping efforts to generate code you trust. Code that doesn't work in a sandbox on Earth probably shouldn't go to Mars. In the text above we had the test stand and the device under test. The device under test should be written to class coding standards. The test stand doesn't have to be written to standards, it won't go in the labs. The device under test could be a few lines of code or it could be a whole function. In case you missed it, all this implies that the prototypes get done first.

In office hours you might be asked to show your prototypes. If you have none you might be told to go prototype the thing you are having trouble with and come back later. If you instead can show that you prototyped a bunch of parts but the combinations doesn't work, you are likely to get help.

Debug Messages

Debugging messages are not counted against the ten line limit, so write them and leave them in your code. Labs 2-4 are required to use the debug.h file shown below. You can change the VERBOSE and GRAPHICS lines. The graders will alter them to grade your labs text and graphical outputs. Your code

should use debug prints to output data essential to understanding that the function is working properly. For example:

```
if(DEBUG)printf("friction_percent: vector length is %0.5lf, becomes
%0.4lf (%0.3lf%%)\n", vlength, v2, 100.0 * percent);
```

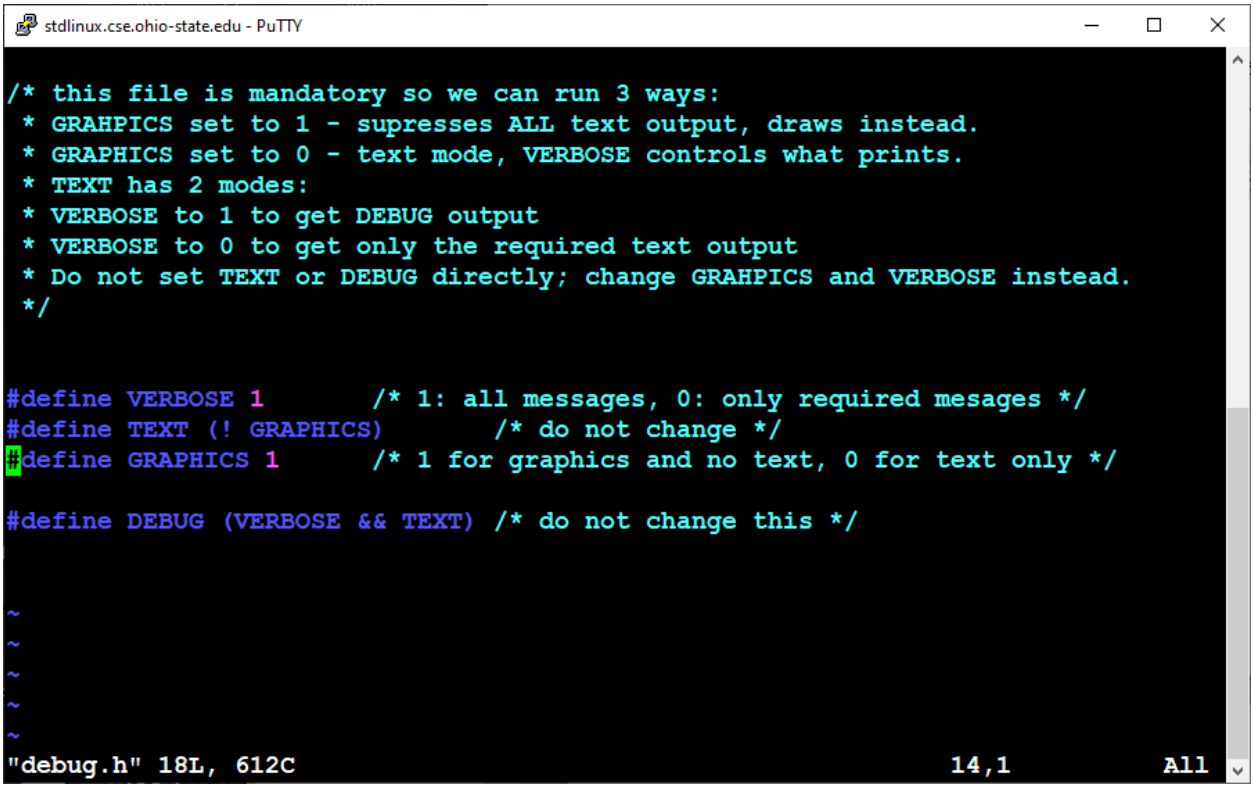
For this class,we have rules about debugging statements.

1. Any if(DEBUG) may only print
2. The print message should start with the function name or ERROR: and the function name.
3. Not code inside a if(DEBUG) construct may change any variable.
4. The target of most if(DEBUG) will be a sinlge printf statement and not a code block using {}
5. The exception is when you need to decide between two different printf statements, see below. These will be rare compare to the single printfs.

```
if(DEBUG)
{
    if(vlength > 0.0)
    {
        printf("friction_percent: vector length is %0.5lf, STOPS.\n",
vlength);
    }
    else
    {
        printf("friction_percent: vector length is %0.5lf, already
stopped.\n", vlength);
      }
}
```

It might be a good idea to write your code one function at a time and test each one. Use a few carefully placed debug statements to see inside. These are really good for displaying mathematical results that you can then check with a calculator. Later on, when testing with real data, you can turn debugging back on if things don't look right. Your code should have some debugging statements onboard. You can expect graders and the instructor to ask you to turn them on when they help you with your code. You can epxect them to ask you to add a few to your code to give better insights as to what is going on.

Here is debug.h:



No Global Variables

The graders have a script that detects all global variables. Global variables are not permitted.

Multiple File Development

Starting with lab 2, there will be multiple .c files in your lab. Organize them by common themes. Some files and the rules about what they hold will be in the lab write-up. The makefile and scripts you got in lab 1 will prove useful. There is a “headers” target in the makefile, allowing you to type “make headers” at the command line.

Makefiles & Submissions

You are responsible for your makefiles and they are required to be submitted. Be sure all compile phases use the ansi, pedantic, and 2 W warning flags along with –g since you will need gdb. Mark any targets that graders should grade. It’s OK to have more than 4 prototypes marked as gradable; they will stop after 4 good ones.

It is a really good idea to have your makefile test your zip file when it builds your zip file. The code in your directory might all be good but if your zip forgets something, you get a zero. Test that zip in an empty folder! Students who don’t test their zips run about a 1 in 100 chance of getting a zero on an otherwise perfect lab.

Once you submit to Carmen, instantly copy the zip out of Carmen to a clean folder and test that it didn't get corrupted on the way into Carmen. Students who do not test that their zip in Carmen is good have something like a 1 in 250 chance of getting a zero on an otherwise perfect lab.

Header Files

The compiler will warn about unknown functions, so you need header files that carry the function prototypes for fuynctions defined in the other files. Much of the data you need for your header files will be found in the .vs files that headers.sh generates for you. If you add a function to a C code file or change the signature of one, do the following **after you save that file**:

1. make headers
2. edit the .h file that matches the .c file you just changed
3. read in / paste in the contents of the matching .vs file over top of the existing function prototypes, keeping the existing comments at the top of the .h file.
4. then issue the make or make labX command – all C files will update and include the changed header(s)

Do not simply copy .vs files over .h files. The .h file needs comments at the top at a minimum that won't be there if you overwrite using the .vs file.

Make Backups

It's a great idea to make backups. If you have version control, use it! But be sure that you also have off-machine backups as well.

The Raw Truth

In this class, students who invest in the up-front activities do better than those who do "all the extra stuff" after their lab code is working. Two signs of poor performance are "I'll put the comments in later," and "I'll do the prototypes after I get the lab working." If you can't write a clear comment about what a function is supposed to do, you might not be able to write clean code to do it. It's OK to change comments if you redesign a function, but they need to match. Labs 3 and 4 are so long that debugging them all at once can easily take longer than you have. Your prototypes let you test small, manageable chunks of high-risk code and turn them into tested components that you trust (or at least that you know how to debug). Keeping your debugging messages on-board means you can recompile and find out what your code was thinking. All these things make you better at designing, coding, and debugging large chunks of software.

Students who do all of the upfront stuff tend to get faster at doing the labs. Students who do it after the fact tend to get buried under the load.