

CSE 2421 Lab 1

**Learning Linux, editing text,
makefiles, and the debugger**

Due: Thursday 18 May 2023 11:59 PM

**Early: 2 bonus points if submitted by Tuesday 16 May 2023
11:59 PM**

Late: Saturday 19 May 2023 11:59 PM

We'll be using Carmen's timestamps to determine lateness or not. If you are not enrolled, you may send me an email with the zip file. If you are unable to submit by the deadline, you need to send me a screenshot of your finished files with the clock visible and then **still** submit to Carmen when you can.

This lab contains three parts. Make sure you do all of them!

Part 0: Logging in and choosing an editor

I'm including this part because I think it's really important to make an informed decision about what sort of text editor you're going to be using for this class. There are lots of options (including vi, gedit, nano, emacs, and more). I recommend either vi or gedit, but primarily vi. What follows is copied from Prof. Kirby:

"Right now you need to pick which editor you will use. In piazza in the General Resources section are two vi tutorials. The original editor was vi. That was updated to vim, but "vi" as a command on stdlinux aliases to vim and that is a good thing. There is also gedit.

There are 2 links to vi tutorials in the General Resources section of the resources page in piazza. The reason I push vi is that once you master it, you are faster than the people who use gedit. Time is one of your most precious resources in this class. Be sure to do most of your editing in edit mode to take advantage of vi's power. (That means hit escape and navigating instead of staying in insert mode all of the time and using only the arrow keys).

Gedit has appreciably no learning curve. It's so easy we know you can look at the interface and figure it out. Gedit is by nature graphical, so you need FastX running or you are doomed.

The vi editor will run in a command window, so no FastX is needed. If you are using putty, vi may be your best choice. I personally develop on 4 putty windows, which fits in and fills a decent sized monitor. I use 1 for running make, another for testing code and 2 have vi to edit code.

Tradeoffs:

- Vi is strictly keyboard biased, so there is no reaching for a mouse. A decent typist in vi edits more quickly because they never take their fingers out of the keyboard. Gedit requires constant mousekeyboard switching.
- Vi will run without an X server, so if you have slow link speeds the low resource demand of PuTTY and vi may be your only choice.
- Vi has some capabilities that gedit doesn't have, like changing the indent level of an entire block of code with a 2 keystroke sequence ">% " instead of multiple lines of mouse / keyboard / tab / click madness.
- Vim color codes text and will color code mismatches on open/close parentheses and braces. • Vi understands tags, allowing you to jump to the definition of a function from any point in the code where the function is called using ^] (control right bracket).
- Gedit is dirt simple but you have to teach yourself vi. If you are using vi and forget that you are in edit mode and not insert mode, your keystrokes all have meanings and the act of typing text will do strange and unpleasant things to your file, especially if you look at your keyboard when you type instead of at the screen. Vim has multiple undo levels so that's not as scary as it used to be. Using gedit means getting something done right now today but being slower all semester. Using vi means teaching yourself one more thing early on and buying extremely precious time for labs 3 and 4."

I also took the liberty of ripping those vi resources he talked about and put them on Carmen. They will come in handy!

Something else you should do before getting started on the lab is creating your readme file. Simply create a file called "readme" and include your name at the top. As you progress through the lab, you may see different parts (labeled things like **Part 1**). When you see this, write it down in your readme. While in a numbered part of a lab, you'll come across various questions, prepended with a bolded Q and a number (such as **Q5**). When you see this, include the Q and the number, along with your answer to the question(s) in the file. A brief example:

Name: Rob LaTour

I certify that I completed all of the work myself with no aid from anyone aside from the instructor or the undergraduate graders.

Part 1:

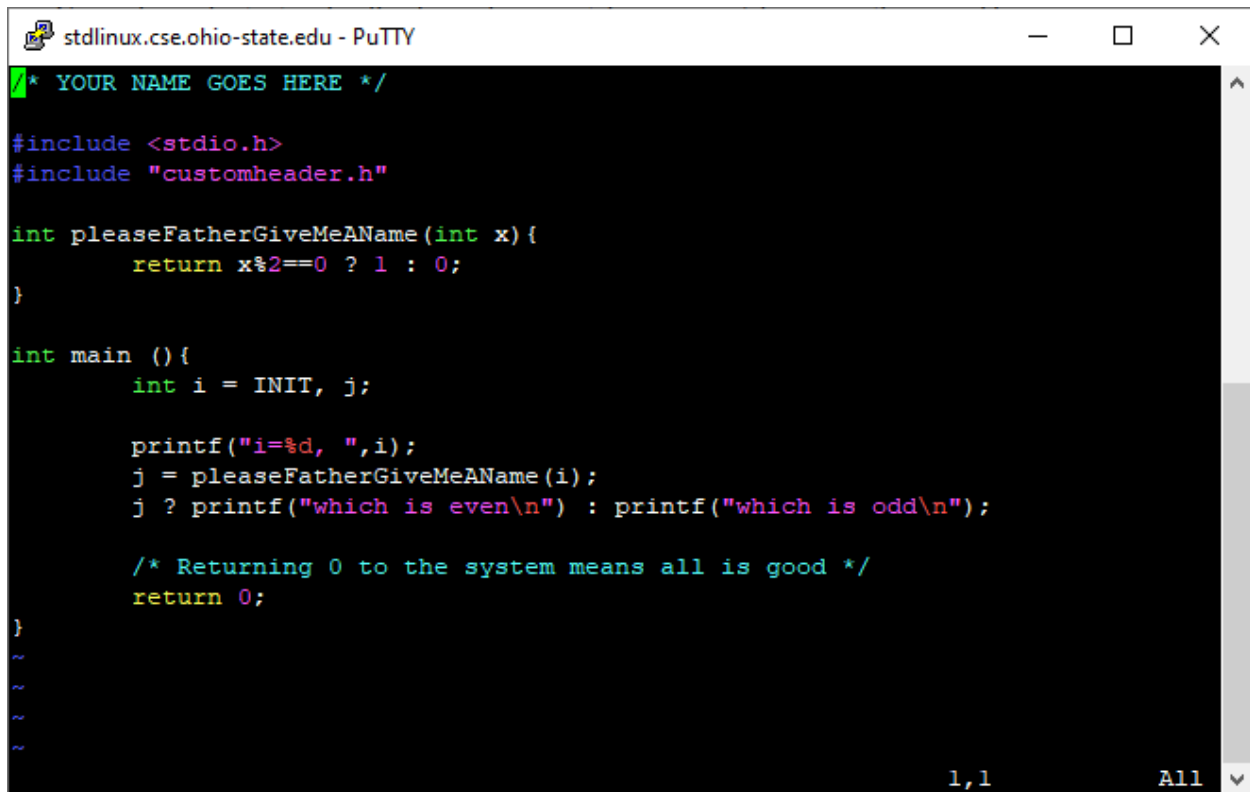
Q1: Don't make fun of me, but my favorite color is gray

Q2: The definitive answer is that Han did shoot first. The original version is the one that captured the heart of the public, and thus the director lost creative control. In this essay I will...

Part 1: A quick and easy C program

Remember not to copy and paste code from slides or lab descriptions, because you'll get different characters from what you're expecting and things will break. You'll need to type this stuff manually!

Create a file called lab1.c and make it look like this:

A screenshot of a PuTTY terminal window titled 'stdlinux.cse.ohio-state.edu - PuTTY'. The terminal shows a C program being edited in vim. The code is as follows:

```
/* YOUR NAME GOES HERE */

#include <stdio.h>
#include "customheader.h"

int pleaseFatherGiveMeAName(int x){
    return x%2==0 ? 1 : 0;
}

int main (){
    int i = INIT, j;

    printf("i=%d, ",i);
    j = pleaseFatherGiveMeAName(i);
    j ? printf("which is even\n") : printf("which is odd\n");

    /* Returning 0 to the system means all is good */
    return 0;
}
~
~
~
~
```

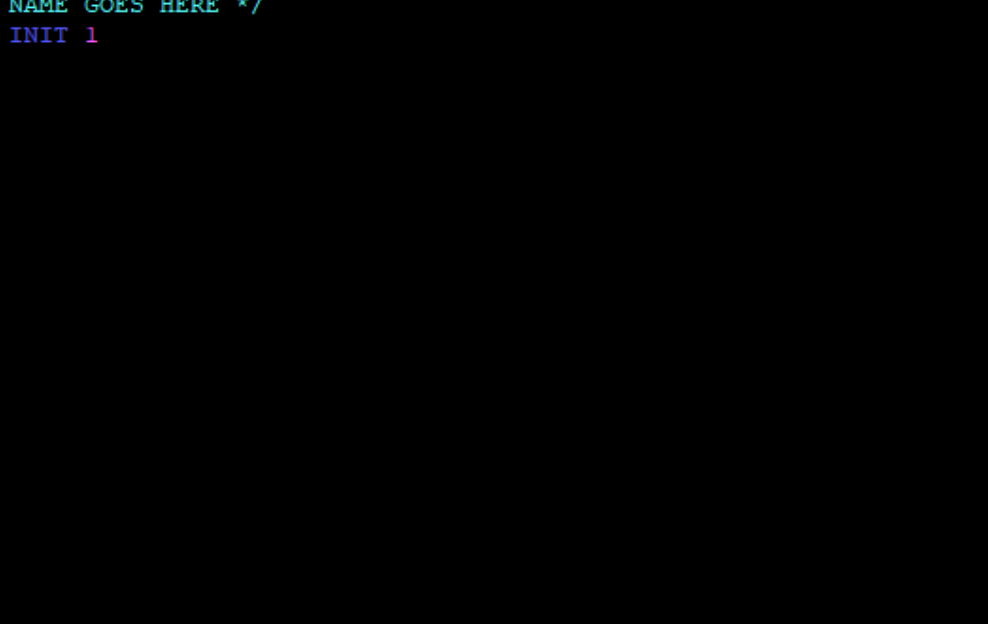
The status bar at the bottom right shows '1,1' and 'All'.

(Side note, you can change the color scheme of vim by using the :color command. This one is called "evening")

Please replace the first line with your name and provide a different name for the function called pleaseFatherGiveMeAName.

Remember to include a newline at the end of every file!

Create a file called `customheader.h` and make it look like this:



The screenshot shows a PuTTY terminal window with the title bar "stdlinux.cse.ohio-state.edu - PuTTY". The terminal has a black background with green text. The first line is a C preprocessor directive: `/* YOUR NAME GOES HERE */`. The second line is another preprocessor directive: `#define INIT 1`. Below this, there are several lines of tilde characters (`~`) representing a file listing or directory structure. At the bottom right of the terminal, the coordinates `1,1` and the text `All` are visible.

Now we're gonna compile this program (this is a one-line command):

```
[latour.2@cse fl1 cse2421]$ gcc -ansi -pedantic -Wimplicit-function-declaration
```

```
-Wreturn-type -g -o lab1 lab1.c
```

Eugh. That's not fun to type out a bunch. Luckily, while at the command line, you can hit the up arrow to bring up commands we've already typed.

Let's talk about what all of those bits of that command mean:

gcc – that's the compiler command

-ansi – tells gcc to use the ANSI (American National Standards Institute) C standard

- pedantic – tells gcc to strictly adhere to that standard

- Wimplicit-function-declaration – warn on implicit declarations

- Wreturn-type – warn about return types

-g – this tells the gcc command to not delete the symbol table so that it can be used by gdb

-o <filename> - this tells gcc to put the executable in a file named <filename>

lab1.c – this is the file that contains the C code that gcc should compile

Both flags that start with W are there because the compiler defaults to being “helpful” (aka not telling you about these things) that can cause problems. All of these flags are required for all of the C code for this class!

After compiling, if there are any errors or warnings, then you need to go and correct those. Resave your lab1.c file after modification and recompile. Once there are no errors or warnings, you can proceed with the next step.

To run the program, simply type the name of the executable file and hit enter:

```
[latour.2@cse fl1 cse2421]$ lab1
```

You should see some output:

```
i=1, which is odd
```

As you’ll note, however, I’m currently in a folder called cse2421. I’d really rather not put all of my lab files all in one place, however, so I’m gonna create a new directory called lab1, then move my .c and .h files into that directory.

Q1: Describe the process by which I can accomplish the above task.

Q2: Suppose I have a group of files in a directory. Describe what the following command would do: `ls -alrt`

Q3: What is the difference between each of the following commands?

```
[latour.2@cse fl1 cse2421]$ cd
```

```
[latour.2@cse fl1 cse2421]$ cd ~
```

```
[latour.2@cse fl1 cse2421]$ cd .
```

```
[latour.2@cse fl1 cse2421]$ cd ..
```

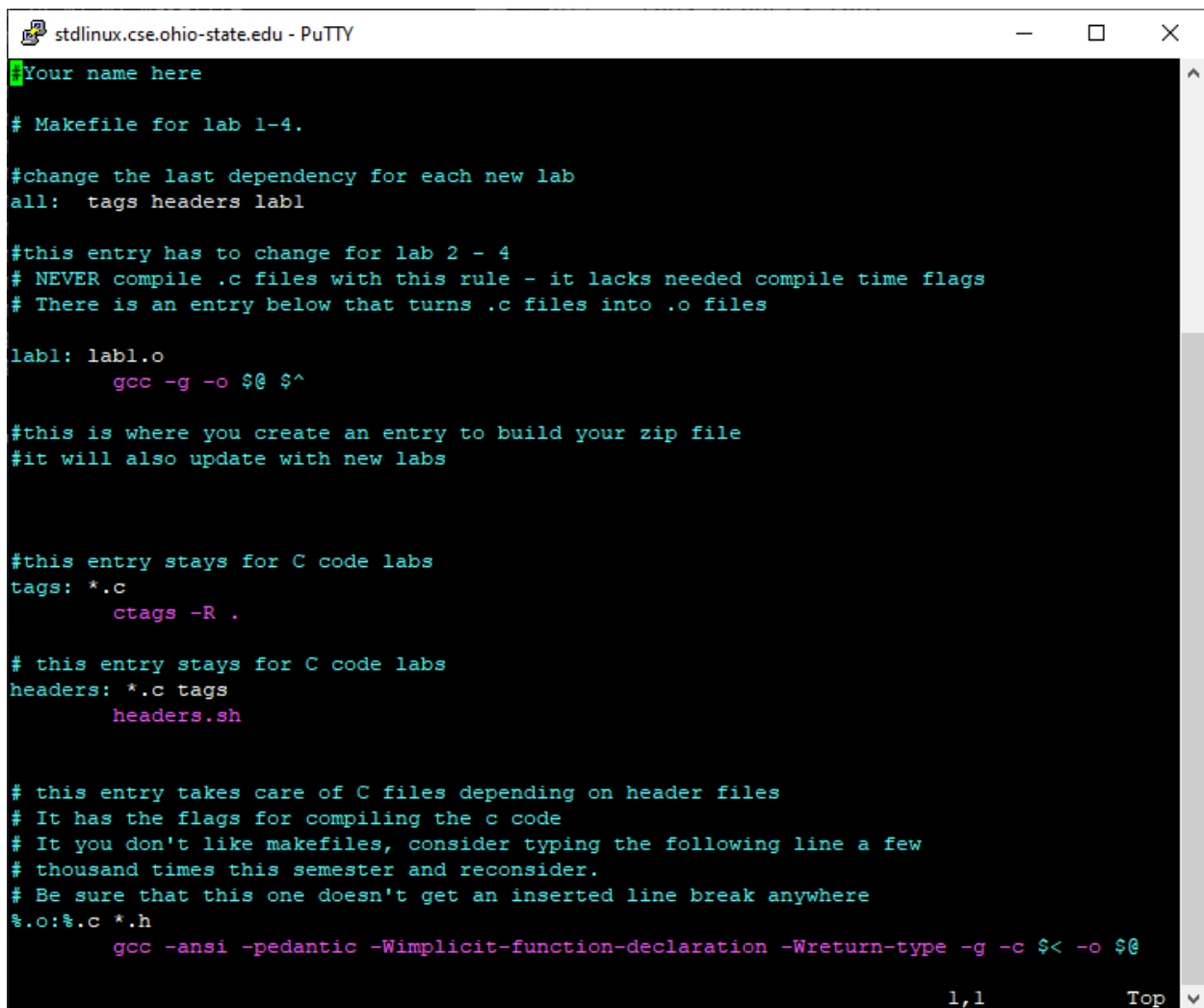
Part 2: Makefiles

Remember that the name of your file must be makefile. Additionally, remember that you *can* (and probably should) include *nix commands as a part of the file. Finally, remember that makefiles are much less flexible than a file containing C source code. Makefiles care about their names, you cannot have more than one in a directory, and they care quite a bit about whitespace.

Posted on Carmen is a zip file that contains a skeleton of a makefile that you'll want to use and modify for labs 1-4 (plus a few other things). You'll want to download this and unzip the contents into your lab1 directory.

If you don't have a graphical way of accessing stdlinux, you'll need to use some sort of Secure File Transfer Protocol (SFTP) to get it from your machine onto stdlinux. FileZilla is great for this, but you can use whatever you'd like.

The skeleton is pictured below, but it's also in the zip file to save you the typing.

A screenshot of a terminal window titled 'stdlinux.cse.ohio-state.edu - PuTTY'. The terminal shows a Makefile skeleton with various comments and rules. The text is as follows:

```
Your name here

# Makefile for lab 1-4.

#change the last dependency for each new lab
all:  tags headers lab1

#this entry has to change for lab 2 - 4
# NEVER compile .c files with this rule - it lacks needed compile time flags
# There is an entry below that turns .c files into .o files

lab1: lab1.o
    gcc -g -o $$ $^

#this is where you create an entry to build your zip file
#it will also update with new labs

#this entry stays for C code labs
tags: *.c
    ctags -R .

# this entry stays for C code labs
headers: *.c tags
    headers.sh

# this entry takes care of C files depending on header files
# It has the flags for compiling the c code
# It you don't like makefiles, consider typing the following line a few
# thousand times this semester and reconsider.
# Be sure that this one doesn't get an inserted line break anywhere
%.o:%.c *.h
    gcc -ansi -pedantic -Wimplicit-function-declaration -Wreturn-type -g -c $< -o $@
```

The terminal window has a scrollbar on the right side, and the status bar at the bottom right shows '1,1' and 'Top'.

You'll need to change a few things in the makefile, but the file does its barebones job in this state.

Any time that you invoke the make command, you'll want to pair it with the -r flag. This forces make not to use any defaults, and those default rules are not the same as what we've been using!

Once you've got all of the files unzipped, go ahead and run the make utility.

Q1: Did anything happen? Should anything have happened?

Try running a different command:

```
[latour.2@cse fl1 lab1]$ touch customheader.h
```

Q2: Did anything happen? Should anything have happened? What if you run make again?

Q3: Add to the makefile a set of entries that zips your source files, header files, readme file, and makefile itself into a zip file. You can base your work off of the snippet provided below:

```
lab4.zip: makefile *.c *.h lab4readme
    zip lab4.zip makefile *.c *.h lab4readme
    # optional below here - remove old install folder
    rm -rf install
    # create the install folder
    mkdir install
    # unzip to the install folder
    unzip lab4.zip -d install
    # make ONLY the lab4 target, not lab4.zip
    make -C install lab4
```

Q4: Add a target to your makefile called “numbers”, make it depend on “makefile”, and give it 3 rules (these are lowercase Ls, “el”):

```
ls -lt >> numbers
date >> numbers
ls -lt >> numbers
```

Issue the following command:

```
[latour.2@cse fl1 lab1]$ make -r numbers
```

Check out what’s in the numbers file, then open your makefile, add and delete a character, save your makefile, and then run the command again.

Q5: What does the numbers file look like now? Why?

Important note regarding makefiles: although you may not share the Lab 1 makefile with your classmates, since it is part of the lab assignment itself, you *may* share makefiles for labs 2, 3, and 4! Feel free to post them on Carmen (or in the unofficial student-led Discord) if you figure out a cool way to make your life easier. Your classmates will certainly appreciate it.

Part 3: Debugging

Let's see if we can figure out how to debug effectively. Firstly, we'll need to make sure that we have an executable file from running our compile command on lab1.c. If our makefile is done correctly, we can simply use the executable it produces for this purpose, since the makefile includes the -g flag.

First, we can launch the debugger with the following command:

```
[latour.2@cse fl1 lab1]$ gdb lab1
```

This launches gdb, which will change your prompt to something that looks like (gdb). We'll want to add a breakpoint to the beginning of main, which we can do with the following command:

```
(gdb) break main
```

You should get a response that looks something like this:

```
Breakpoint 1 at 0x40054c: file lab1.c, line 11.
```

Now we are ready to begin walking through the program. All we have to do is enter the command to begin running:

```
(gdb) run
```

From here, let's walk through each line and see what happens as we go. We're going to use the following command over and over again until the program terminates (you'll know you're done when it says that the process exited normally):

```
(gdb) next
```

Q1: As we ran the next command, lines (and their line numbers) were printed to the screen. Is that the line that just ran or the next line that is going to be run? How do you know?

Let's walk through the code again, but let's play with some more things. Without quitting gdb, go ahead and run the program once more.

We're gonna enter the following commands, rapid-fire style!

```
(gdb) print i
```

```
(gdb) next
```

```
(gdb) print i
```

```
(gdb) set variable i = 5
```

```
(gdb) continue
```

Q2: What... what just happened? What was the value of i at first? What was it the next time we printed it? Why? What about at the end? Why? Did the output change? Why?

Q3: What if I wanted to step into the function? How would I do that?

Play around with various debugger commands to feel comfortable with what they do and what output you're seeing. Then, we're going to quit the debugger and relaunch gdb. This time, set the breakpoint to your function, rather than main. Start running the program, then modify the value of x to be 32. Finally, allow the debugger to finish its run.

Q4: Did the output change? If so, how did it change and why? If not, why not?

Next up, we're going to test out a provided file called proCoder.c. Compile this (or better yet, create an additional target in your makefile and run `make -r proCoder`). Did it crash? Great!

Let's figure out what happened quickly, without having to stare at the code for a long time. Toss proCoder into gdb and just run it without placing any breakpoints.

Aha! You should spot that there is now a line that you're given that caused the crash! Let's take a peek at that line in the file (either turn on line numbers in gedit or use vi's power to hop to a specific line. You wanna go to line 17? Just enter `":17"` in vi and boom you're there).

Hmm, it's nice to know what line caused the crash, but it would also be good to know how we got here. Luckily, there's another command we can use to give us that information:

```
(gdb) backtrace
```

This should provide for us a list of calls that led to our crash, with each being labeled by a number at the start, including function name and parameter values. These entries are called stack frames, and they are amazing.

Just by looking at the backtrace, you can probably guess what the problem was, but we can actually use stack frames to figure out exact variable values when these functions were called. Enter the following command to "zoom in" on one of the frames:

```
(gdb) frame 1
```

Now we can use the print command to see what the local variables are at this layer within the program, so we can see what the values of bot and i are. While you're peeking at a frame, you can also use the up command or the down command to swap between frames.

Q5: What line caused the crash in proCoder? Why?

Q6: Based on this, what change would you need to make to fix the proCoder.c file? Answer this part honestly: was using gdb faster than it would have been to stare at the code to try to figure out what was happening?

Part the last: Extra information and submitting

Professor Kirby is very kind and wrote a couple of files included in the `makefileresources.zip` that perform some pretty interesting functions. Here's what he has to say about those:

"Shell Scripts and AWK Scripts

Running the `make` command on the `all` target invokes the `ctags` program and the shell script `"headers.sh"` (which invokes `awk` using the `"headers.awk"` program. Along the way these will generate various files:

- `tags` is the output of `ctags`
- `headers` is updated when `headers.sh` runs
- `lab1.vs` is the file created from running `headers.awk` on the `tags` file

Use the `cat` command to look at the contents of each of these files.

```
[kirby.249@cse-sl6 lab1]$ cat tags
```

```
[kirby.249@cse-sl6 lab1]$ cat headers
```

```
[kirby.249@cse-sl6 lab1]$ cat lab1.vs
```

The `tags` file is used by `vi` to allow it to jump to the right file and the right line number in the file when you tell it to go to a function definition. The `headers` file only exists to keep `make` happy so that `make` has a target that it can check the file touch date on. The interesting file is `lab1.vs`, which stands for "visible symbols." I created this script to make life easier. If you need to create a header file – and you will in labs 2-4 – then this file has the exact text you need for the header file."

As you know, you must submit your lab to Carmen in a zip file. You should include in that zip file your C files and header files (in this case only `lab1.c` and `customheader.h`), your `makefile`, and your `readme` file.

While you could use the `zip` command to create your file, I cannot stress enough how good of an idea it is to put the `zip` command in your `makefile` (as requested of you in Part 2 of this lab). If that is included, then every time you update your files, you can simply run `make` and you not only have compiled files but an up-to-date zip file as well.