Lab 1

Version 2

V1: The supplied lab4.zip sample makefile is extensively reworked.
V2: All references to anis are fixed (2 different places with command lines). Also took out implicit function and put in format. (End of page 15 and start of page 16.)

Contents

Dates

Early: 2 Bonus Points (i.e.10% bonus) awarded if correctly submitted by Sun 15-Jan-2023, 11:58 PM

Due: Tuesday 17-Jan-2023, 11:58 PM

Late until 11:58 PM on Wed 18-Jan-2023 (anything with a Thursday date or later is not accepted)

Determination of whether or not you have met the due date will be via the timestamp indicated within Carmen. If you are not yet enrolled, you may email me the zip file by the dates above. If you **are** enrolled and send me a zip file, I am likely to ignore it and you get a zero.

Introduction

A determined student can get through this in about an hour. For once you have plenty of time for a lab. Almost everything this lab asks you to do is there to give you the survival skills you need to be effective in doing

labs in this class. Take your time. Savor this one. This is the easiest 22 points *ever* in a lab in this class. **Your goal should be to maximize what you get out of doing this assignment.** For everything you do, ask yourself "why?" and "could I do that from memory?" and "what are these commands telling me?" Become expert makefile editors. Pwn gdb and make it your faithful little dog walking obediently at your heel, ready for any of the many commands in your bag of tricks. Become "ripping fast" with your editor.

There is an appendix on commands at the end of this writeup that might be useful if the writeup tells you to do things that you don't know how to do. Google can be massively helpful here; "How do I add execute permission to a file in Linux" will correctly take you to writeups on how to use the chmod command.

Anything marked with a checkmark in this writeup is something you should do as part of doing the lab.

There are questions marked with numbers – your answers to those questions go in your README file. Skip ahead to the questions section and make a copy of the questions so that you can answer them as you go.

Get Connected First and Create Your Directories

This writeup assumes that you can get connected to stdlinux and are looking at one or more windows showing you a command prompt. If you are having trouble, see the section at the end of the writeup with details on getting connected.

- ✓ Create a cse2421 directory
- ✓ Change to that directory
- ✓ Create a lab1 directory in the cse2421 directory you just created
- ✓ Change to that directory. All work on lab 1 will be done here

You need to be able to move files from piazza to stdlinux. In the general Resources section of piazza is an article "Top 7 Free SFTP or SCP Clients for Windows" that might be worth reading. My personal favorite is FileZilla, which you can get from https://filezilla-project.org/ for free. Clicking files in the resources area of piazza will get them downloaded to your PC, but it will take FileZilla or equivalent to get them to or from stdlinux.

- ✓ Transfer buildfiles.zip and zd.c from the Lab 1 area in piazza to your lab 1 directory on stdlinux
- ✓ If you are going to use vi / vim and you don't already have a .vimrc file in your Linux home directory, transfer the file vimrc from the General Resources to your home directory. Then rename it .vimrc (files that begin with . are not listed by ls unless you use the -a flag). If you are not going to use vi / vim on stdliunx, skip everything in this check list item. The command **mv vimrc .vimrc** should do the trick to rename the file.
- ✓ If you use vi from xterm instead of PuTTY, change the color scheme in the vimrc file from evening to default
- ✓ Use the unzip command to unzip buildfiles.zip in your lab 1 directory. You simply type **unzip buildfiles.zip** at the command line.

There are tons of awesome in the three files:

- There is a sample **makefile** that has the one rule to build all C code files. It has other build targets as well that are there to make your life easier. Notably it has the headers and tags targets.

- The file **headers.awk** is an AWK program. AWK is a "little language" that can be very useful. You don't have to mess with it and you should not change it. If you are looking to learn more, go learn the language; you can master the basics in under an hour.
- The shell script **headers.sh** is a collection of bash commands. The original was written by Neil Kirby and extended by Kyle Rosenberg in SP22. In AU22 I improved to its current, more awesome form. This shell script will automatically build your header files for you and keep them up to date.

You do need to do a few things to some of those files:

- ✓ Run the **ls –l** command after you have unzipped the files. Pay attention to two things. The file modification date is shown in long form of ls. It also shows the permissions, which use – r w x to indicate read, write, execute permissions, and – to show the lack of a particular permission. Look at headers.sh and note that it doesn't have write permission. We need to modify the file.
- ✓ Execute the command **chmod +w headers.sh** to give write permission to that file.
- ✓ Run **ls –l** again. Note the changes to the modification date and the permissions on headers.sh.
- ✓ Change headers.sh so that it adds *your* name to your header files. If it were me, I'd change the string **"I AM GETTING A 0 ON THIS LAB"** to **"Neil Kirby"** being sure to leave the double quotes surrounding the correct name. *In all labs, your name is required in your source code files or you get a zero on the lab.* This includes header files (.h), C code files (.c), assembler code files (.s), README files and your makefile. Feel free to read the rest of the shell script and ask yourself if you have time to learn yet another programming language. *If your name has a ' in it, don't put that in! For example O'Brian has a single quote and that will screw things up.*
- ✓ This shell script needs to have execute permissions. Use **ls -l** to see if the file has execute permissions. If permission got screwed up in the transfer process use the command **chmod +x headers.sh** at the command line.
- ✓ The headers.sh file is complex and we don't want it changed once we have the name in there correctly. Run **chmod –w** to remove write permissions on the file. This will stop most attempts to modify the file.
- ✓ The makefile needs your name at the top in a comment as well. Forgetting this one is the leading cause of getting a zero on lab 1. Feel free to read over this gem and become one with what it does. It has comments telling you what to mess with and what to leave alone. *You are responsible for your makefiles.*

Makefiles in lab1 must **not** be shared, they are part of the assignment. From lab 2 on, makefiles may be publicly shared as a public service.

Let's Build Some C Code!

You need to pick an editor. If you are going to use vi / vim and haven't used it before, now would be a good time to fire up one or both of the tutorials mentioned in the piazza General Resources area. They are **A vi tutorial** and **Interactive vim tutorial**.

The original Unix screen 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.

The reason I push vi is that once you master it, you are faster the people who use gedit. **Time is one of your most precious resources in this class**. Be sure to do most of you 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).

There are a **lot** of keyboard shortcuts in vi and you never ever stop to reach for the mouse. The tutorials are a good introduction. You don't have to use vi (or vim the modern equivalent), but it will be there when X-windows is not.

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. Recall the day 1 lecture – sometimes FastX isn't working when PuTTY is working.

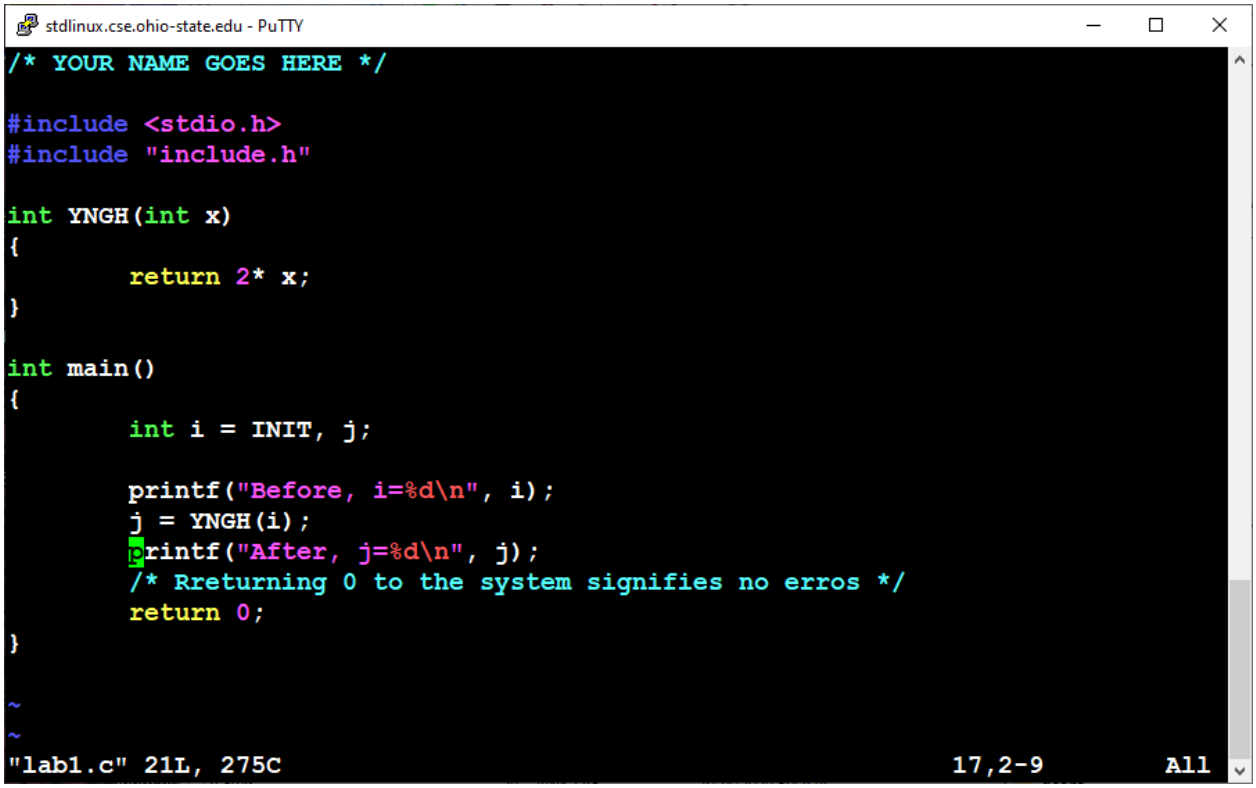
The vi editor will run in a command window, so no FastX is needed. If you are using PuTTY, vi may be your best choice. I personally develop on 4 PuTTY windows, which fits in and fills a decent sized monitor. I use 1 for running make, another for testing code and 2 have vi. One vi window is "where I am making edits," in other words, the file you are actively editing. The other vi window is to look at stuff you need to see in order to be able to do editing in the main editing window. For example, you may be editing code that uses a struct defined in a header file. You use vi to view the header file in order to be able to edit the code file.

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 mouse-keyboard switching. That task switch is more than physical motion time, it's a tiny task switch in your brain.
- 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 gedits 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.

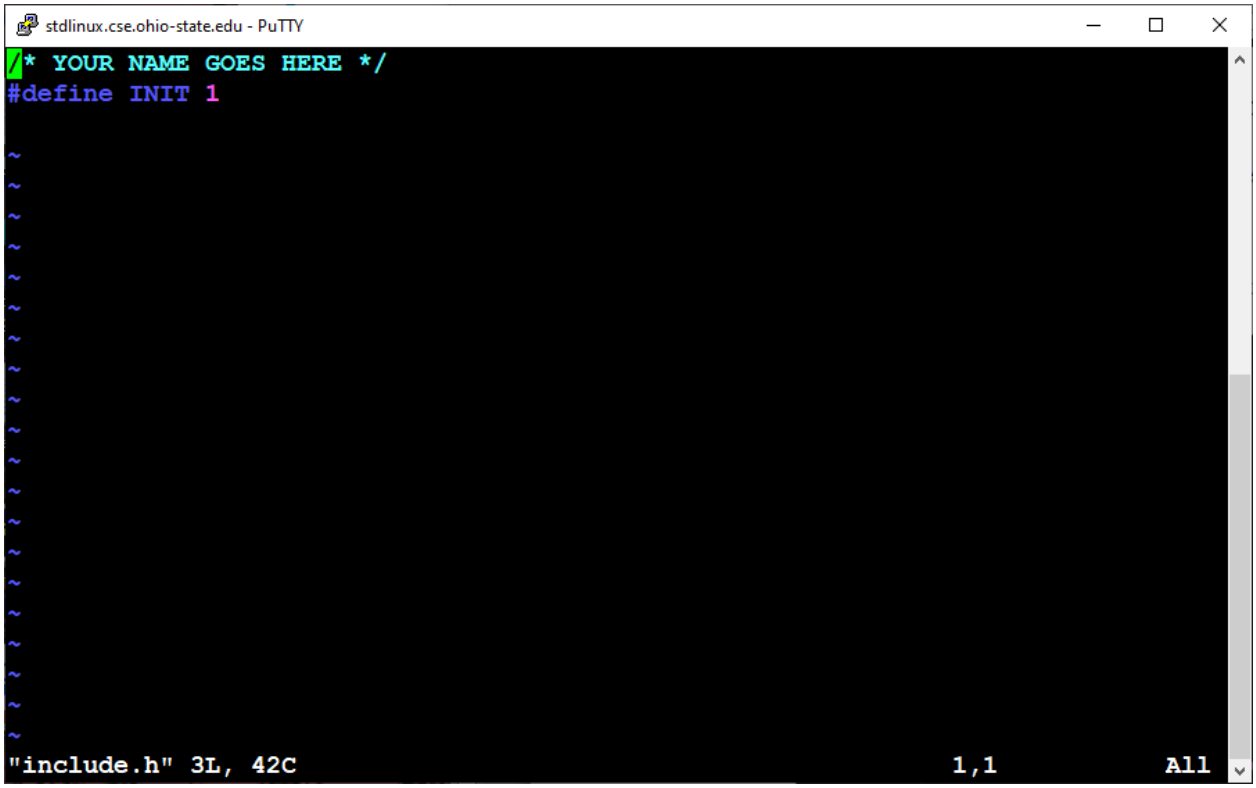
Vim color codes text and will color code mismatches on open/close parentheses and braces., see picture below:

Using PuTTY and vi gives you awesome color coding:



Time to get back to work:

- ✓ Using one of the available text editors, start editing a file named **lab1.c**
- ✓ Type the previously shown program as it appears above. You should be sure to insert your name where indicated. It's OK to fix typos in the comments.
- ✓ **Be sure to hit the enter key after the last curly bracket.** The indentation is important for readability and is a standard that you should always incorporate when writing your programs – use the tab key to indent for the first indentation level and four spaces more for each additional level.
- ✓ Change YNGH to some other appropriate function name. Save the file.
- ✓ Now edit a file named **include.h** as shown below. Be sure to change it to include your name. This kind of header file is not built automatically by headers.sh, it's one the programmer creates.



Once all of these files have been saved, it's time to build some things. First we will start playing with the makefile

- ✓ Issue the command **make -r all** to build the lab1 executable along with tags and headers

Take a look at the makefile and see if you can tell what is going on, at least in general. The tags target ought to build without any issues. The headers tag will build correctly unless you botched changing your name in the headers.sh file. If that happens, unzip a fresh copy and edit it again.

If your lab1 target failed to build, fix any syntax errors until it does build. Note the YNGH "Your function Name Goes Here" is in there twice; you need to have changed it in both places to get it to work.

- ✓ Issue the following command at the prompt:
  gcc -std=c99 -pedantic -Wformat -Wreturn-type –g -o lab1 lab1.c

If that command failed, try it again, only this time actually type the whole thing out instead of using copy / paste.

- ✓ Run the lab1 executable by issuing the command **lab1** at the command line

You should see the output (or something similar)

Before, i=1
After, j=2

on the screen.

Header Files and the Tools that Build Them (AWK and Shell)

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]
- lab1.h is created by headers.sh

Let's make sure that those files are present and up-to-date.

- ✓ Give the command **make –r** all at the command line

Use the cat command to look at the contents of each of these files. There are questions in the questions section about some of this stuff.

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

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. You do that by putting the cursor on a the first character of a function name and hitting control-closing bracket "**^]**" to ask vi to take you there.

- ✓ If you are using vi, edit the lab1.h file and move your cursor to the m in main. Hit control closing square bracket. If vi thinks there is more than one place for this symbol, it will give you a numbered list to pick from. Otherwise it jumps to the place, possible in another file. Vi can only do this if the tags exist and are current. Exit the editor.

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 first really interesting file is lab1.vs, which stands for "visible symbols." I created this script to make life easier. The information in the vs file is what is needed to create a header file needed by other files when their code calls the functions in our C code file. But those declarations need to be in a header file

Far more interesting is lab1.h, the header file listing all of the functions in lab1.c that can be called by other files. Without automation like these scripts, header files have to be created by hand.

You need to be careful about what you name header files. The headers.sh script will overwrite anything you already have in a header file named with the same stem as a C code file. (In this case lab1.h and lab1.c have the same stem.) *When you need a header file that contains #define symbols, give it a name that is different than any C code source file!* This is why include.h is named the way it is.

Header files make the compiler happy when code in one file calls routines defined in another file. We saw this in the slides on makefiles. The declarations in the header files ***must*** exactly match the declarations provided by the function definitions. If they don't you get impossible to find run-time errors that are extremely hard to detect by reading the source code.

- ✓ Add the following 2 lines of C code to lab1.c below the #inlcude lines and above any other functions and then save the file.

```
double foo(double x)
{return 2.0 * x; }
```

- ✓ Run **make -r all** and note that the script updated the lab1.h header and that make rebuilt lab1.
- ✓ Issue the command **touch lab1.c** to change its file modification date without changing its contents.
- ✓ Run **make -r all** again and note that the header did not update but make did rebuild the executable.

Let's change the code and show how we can get the compiler to validate our header files.

- ✓ Add the following line to lab1.c, just below the #include "include.h" line. (Don't copy-paste; Word messed with the double quotes.)

```
#include "lab1.h"
```

Save the file. Your life is a ton easier if you have more than one window up. Compile in one, edit in the other.

- ✓ Rebuild with **make -r lab1** and note that it recompiled. Since we did not get any errors, the definitions in the C code file exactly match the declarations given in the header file. *(Do not make the target all, make the target lab1!)*
- ✓ In lab1.c, change the function foo to:

```
int foo(int x)
{return 2 * x; }
```

- ✓ Save the changes and then rebuild with **make -r lab1** and observe the result. *(Do not make the target all, make the target lab1!)*

Our edit changed the signature of the function foo given by its definition. We did not rebuild the headers, so the old lab1.h header file with the old signature was left alone. The signatures of foo do not match and the compiler gave an error.

- ✓ Run **make -r all** and note that now it builds.

The scripts rebuilt the header file from the C code and now the declaration in the header file match the definition in the C code file. This means that all other files that call this function will fail to compile until you change them to match the new function signature, which is what you want when you change the signature of a function.

It is mandatory in this class that any C code source file #include its own header file in order to validate its header file.

This means that lab1.c must #include the lab1.h file. The best place to put that #include is at the end of the list of all #include in the file. Please order your #include directives as follows:

1. All #include that use <> delimiters go first. These are system files and chances are really good that they have no bugs in them
2. Next are your own header files that have macros in them (#define symbols)
3. Next are your own header files that have struct declarations in them

4. Last are all of your own header files that contain your function declarations, such as lab1.h

Let's Work with GDB

As far as commands you are supposed to execute, this section has shell commands and gdb commands shown in bold after a prompt. It also has a few commands to run marked with a check mark. The shell and gdb commands are in bold and they are indented. Some of this is so we can tell gdb commands apart from shell commands.

Now go to the cse2421/lab1 directory and recompile your lab1.c file. Then use the command:

[kirby.249@fl1 lab1] $ **gdb lab1**

You will enter the debugger running the lab1 program. You will see a new prompt at which you should enter a new command:

(gdb) **break main**

This command will set a breakpoint at the beginning of your lab1 program. You should see a response similar to the one below:

Breakpoint 1 at 0x400543: file lab1.c, line 13.

(gdb)

Now we want to begin running the program so issue the run command:

(gdb) **run**

You should see some output that tells you about hitting the breakpoint we set above.

We can now run the program instruction by instruction by using the **next** command. **IMPORTANT**: Each time you use the **next** command, a line of code will be printed. This is the line of code that will be executed **next**; it has not yet executed; only the lines of code prior to the one just printed have executed.

*A brief note about **step**, **next**, and **finish** (don't execute these commands just yet):*

- *The **step** command walks you through any functions that are called, line by line. This is a pain if your code calls printf! This is helpful if you need to debug the function being called.*
- *The **next** command treats functions calls as a single line of code. The function is executed without stopping and when it is done gdb then waits for input from you. This is for when you don't want to debug the functions being called, such as printf.*
- *The **finish** command tells gdb to complete the current function and then wait for you once it returns.*

These three command are now fair game on a test, so it is a good idea to learn them. From lab 3 on they will save you precious time.

The program will stop at the declaration for i. Then say:

(gdb) **print i**

to see what value i contains. You will want to record these values for completion of questions later.

(gdb) **step**

And then

(gdb) **print i**

Note that it was initialized.

(gdb) **step**

This takes us into printf code, and we don't want to see how much code it takes to print things.

(gdb) **finish**

That lets printf execute until it is done and then gdb stops when printf returns and we get back to our code. Finish really means "finish this function (unless I hit a breakpoint)."

- ✓ Now, use the command, **continue,** to allow the program to complete. Do not exit gdb.
- ✓ Execute the **run** command again to restart the program.

It should stop at your breakpoint again. We use *next* and not *step* this time so we don't enter functions.

(gdb) **next**

(gdb) **print i**

(gdb) **set variable i = 6**

(gdb) **print i**

(gdb) **continue**

Allow the program to complete. Note the output and how it may or may not have changed from the first time it ran. The continue command means "continue executing until the program terminates or hits a breakpoint."

- ✓ Without exiting gdb execute the **run** command yet again.

It should stop at your breakpoint.

- ✓ Step through the entire program using step for your code and next for printf. Feel free to print variables along the way and practice changing them.

Now exit the debugger:

(gdb) **quit**

Now we are going to use our commands another time:

- ✓ Run gdb on lab1 again but set the breakpoint to whatever function name you renamed YNGH.
- ✓ Then issue the **run** command.

- ✓ When the program breaks, change the value of the variable x to 200.
- ✓ Issue the **finish** command.
- ✓ Print the value of i. (See the question list)
- ✓ Issue the **continue** command.
- ✓ Then quit the debugger with **quit**.

Advanced gdb – Stack Frames

For this part, it is up to you if you type the gcc command with all of those flags or you create an additional target in the makefile.

- ✓ Download the **zd.c** file from the Lab 1 section of piazza into your lab 1 folder.
- ✓ Build a **zd** executable, using **make –r zd** after making appropriate changes to your makefile.
- ✓ Run your zd executable on the command line. It should crash.

Don't bother trying to figure out where it crashed.

- ✓ Type **gdb zd** at the command line to bring gdb back into play. Don't bother setting a breakpoint.

When airliners go down, the first thing investigators want to know is **where** it went down.

- ✓ So just type the **run** command to gdb and have the program crash under gdb.

The message you get will tell you what line of what file caused the crash. One of the next things accident investigators ask is "how did the aircraft get here?"

Unless your program has done extremely rude things to the runtime stack, which can happen in this class, gdb will be able to tell us how we got here.

- ✓ Type **backtrace** at the gdb prompt.

What you get is the calling history along with values of parameters for each call. The history includes file names and line numbers.

[Sidenote: If you are using vi, when you are in edit mode you can type **:23** and press enter to jump to line 23 of the current file. If you use gedit, turn on line numbers.]

At the beginning of each line in the backtrace is a frame number. We will deal with how these stack frames got there in the second half of the semester, for now we are interested in the data in each frame. You should see 3 frames, 0, 1, and 2. Higher numbers are closer to main and lower numbers are things that got called. The divide function where it crashed is frame 0. Stack frames are where local variables are stored. We can see parameters like x and y in the backtrace and we can see local variables for the current frame. We can't see local variables like i and j in other frames at the moment.

- ✓ Type **frame 1** at the gdb command line.

This turns gdb's attention to that frame, which is the frame for the function quotient_table where it calls the function divide. Now we can see local variables that belong to this invocation of quotient_table.

- ✓ Use the **print** command to print i and j to see what they were when this call got made.

You could rightly say that we have a really good idea of what i and j hold based on x and y, but the point is we are able to see these local variables if we change frames.

- ✓ Use the print command to see what the value of p is, because we have no other direct visibility into that value. (This value is needed for one of the questions).

If you want to see locals in main, use the **up** command to move "up" the calling chain to higher numbered stack frames. Up takes you towards main. Likewise the **down** command goes "down" the calling chain to lower numbered stack frames.

You are expected to run gdb on any program that crashes prior to asking anyone else for help. If I had a flashing neon font with sound effects I would use it for that last sentence. If a grader asks what gdb told you and you didn't run it, they can say "come back after you run gdb." That could cost you your place in line and now the seven people behind you in line are now seven people in front of you in line.

You are expected to be quick on the draw with the **backtrace, frame,** and **print** commands in gdb. The **backtrace** command alone solves the "how did we get here?" question most of the time. The other two gives the answer to the question, "What were you thinking?" about any function in the calling chain. *It is easy to forget these things between now and when your lab will depend on them.*

Let's Work on the makefile and Build a Zip File

Once you have the makefile in place along with the other files from the lab1.zip file do the following:

```
[kirby.249@cse-sl6 lab1]$ make -r
make: Nothing to be done for `all'.
[kirby.249@cse-sl6 lab1]$ touch include.h
[kirby.249@cse-sl6 lab1]$ make -r
```

There is a question about the above behavior. If make did some building on the *first* make call above, do the three steps above again and it should say nothing to do the first time and it should build the second time.

Let's do some basics with our makefile:

- ✓ Edit the makefile in your lab1 directory.
- ✓ Create a target in that makefile called "stuff" and make it depend on "makefile" and give it the following 2 rules to build it:

  ```
  date > stuff
  ls –lt >> stuff
  ```

- ✓ Write the file and then issue the command "make -r stuff" and see what happens.
- ✓ Use the cat command to see what got put in the file "stuff."
- ✓ Use the ls –lt command to see modification dates.
- ✓ Issue the command "make -r stuff" again. How did it behave differently?
- ✓ Bring the makefile up in an editor and save it.
- ✓ Issue the "make stuff" command. Note how it behaved this time – why did it do so?

The makefile already has code to build the lab1 executable.

- ✓ Now issue the command "make –r lab1" and see what happens. If nothing happens, edit and save lab1.c and try again.
- ✓ Then issue the command a second time. Isn't "make –r lab1" so much easier than the command line that does the actual work of compiling the code?

Even though you were given this makefile, you are responsible for getting it right. You are responsible for making it do what you want it to do. You will need to add a target that builds your zip file. You will add a few simple targets as outlined below.

For all labs, the zips files must be built via make and not by calling zip on the command line.

Study the makefile fragment below. Identify target(s), dependencies, and rules. Every lab submission we turn in will be a zip file. Get in the habit now of using makefiles to build your code and to build your zip file – it is required. For lab 1, we will have different dependencies and a different target for the zip file than what is shown below. But you should customize this code and use it in every lab. This particular set of rules does more than create a zip file, it tests that the zipfile can be unzipped and the most important target can be cleanly compiled. In other words, the extra rules are there to prevent a zero score on an otherwise working lab. **Your lab will be counted as <u>late</u> if the makefile does not self-test the zip target**! (I'm tired of hearing students whine about labs that get a zero because the zip file didn't have everything it needed to have.) This also means that your lab is late if you hand-built the zip file.

Make sure that the rules in your makefile use tab and not spaces!

```
# Comments in a makefile start with sharp – and are mandatory
# do not forget your makefile and your readme in your zip file

lab4.zip:  makefile *.c *.h README_LAB4
      # in the rule below, $^ can save you – why?
      zip $@ $^
      # self-test is required: first 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 all other things that get graded too (never the zip!)
      # the above means things like stuff and all prototypes
      # LOOK FOR ERRORS BELOW HERE
      make –C install -r lab4
      make –C install –r p1
      # LOOK FOR BUILD ERRORS ABOVE HERE
      # Finally: remove install folder so we don't accidentally use it
      rm –rf install
```

- ✓ Edit your makefile. Using the above code as a guide, give it the capability to create and test your zip file.
- ✓ Use **make -r lab1.zip** to prove that a zip file gets created and that the self-test works.

Be sure to turn in your makefile along with all of the other files that make up lab 1. All labs will be built with makefiles and will require a makefile to be turned in. **Comments are mandatory in all makefiles!**

Note: The customizations you make to your makefile **cannot** be shared in lab1, because they are part of the programming assignment. From lab 2 on – but not lab 1 – makefiles **can** be shared. Do this on piazza to gain the admiration of your peers when you find some new thing that makes life easier for you. Or you can answer questions about makefiles – from lab 2 on – to help people out.

The Question List

These questions and their answers go in your README file.

1. Within gdb, the value of the variable i before the initialization is:

2. Within gdb, the value of the variable i after the initialization is:

3. What does touch do and why did it change the behavior of make?

4. In gdb, you walked through the code using next and using step – describe the differences in their behavior:

5. In gdb, what did finish do?

6. On the final run, after you set the value of x to 200, what was the

value of i back in the calling function?

7. What kind of recognizable things were found in the tags file?

8. How do the contents of the .vs file compare to your C code?

9. Give the line number and filename for where the zd executable crashed.

10. What was the value of p in zd when zd crashed?

11. What did the backtrace command show you in gdb?

12. What are you expected to do before asking anyone for help with a run-time crash of your program?

Submission

All labs must have a READ_ME file. Edit a new file in the lab1 directory called README_LAB1. The filename must be exactly as stated here. Any other filename will not be considered correct and the content will not be graded. Inside this file put what is below, after editing the fields marked with <>:

THIS IS THE README FILE FOR LAB 1.

<your name goes here>
BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE PERFORMED ALL
OF THE WORK TO DETERMINE THE ANSWERS FOUND WITHIN THIS FILE MYSELF WITH
NO ASSISTANCE FROM ANY PERSON OTHER THAN THE INSTRUCTOR OF THIS COURSE
OR ONE OF OUR UNDERGRADUATE GRADERS.

<the questions and answers go here>

Note

The easiest way to upload files specific to the stdlinux environment is by opening Carmen from stdlinux and uploading the corresponding .zip file from there. To open Carmen from stdlinux:

i) login to stdlinux using CSE remote access.
ii) Open terminal
iii) Type "firefox &". Click **enter**. A new window will open with firefox web browser in it. NOTE: firefox is the browser recommended by Carmen administration. If you are an experienced Linux user, you may use another browser at your own risk.
iv) Navigate to Carmen.osu.edu from that window.
v) Use given links to upload assignment to Carmen.

- Your programs MUST be submitted in source code form. Make sure that you zip all the required .c files for the current lab (and .h files when necessary), and any other files specified in the assignment description. Do NOT submit the object files (.o) and/or the executable. The grader will not use executables that you submit anyway. She or he will always build/compile your code using
  gcc -std=c99 –pedantic Wformat -Wreturn-type -g,
  and run the executable generated by that command.

The other way to submit a file is to use Filezilla or any of the other decent file transfer methods to pull your zip file to your machine and then use your regular browser to submit that to Carmen. See the link about Top 7 Free SFTP or SCP

Clients on piazza in the General Resources area.

- It is YOUR responsibility to make sure your code can compile and run onCSE department server **stdlinux.cse.ohio-state.edu,** using **gcc -std=c99 -pedantic -Wformat -Wreturn-type –g** without generating any errors or warnings or segmentation faults, etc. Any program that generates errors or warnings when compiled or does not run without system errors will receive 0 points. No exceptions!

Appendix of Useful Commands

You can log on to stdlinux remotely, as explained in class, using the remote access method appropriate for your system, or you can go to a CSE lab, as mentioned above. The way to access stdlinux from a lab is explained below.

After you have logged on to Windows and then Linux, as explained above, to get a terminal window (if you are in graphical mode), from a computer in a lab, choose the "Applications" tab (top left menu system), then from the drop down menus, choose "Other" then "Terminal". Notice the picture of the terminal. This icon may also be to the right of your 3 tab main menu of "Applications", "Places" and "System".

A window pops up with your "home directory" name. For example:

[kirby.249@fl1 ~] $

You are ready to enter LINUX commands. Thus, this is called the "command line prompt". To change your password, you have to change your OSU name.nnnn account password.

Below are some other good/common commands for you to learn. The first set of characters up until the first space is the command name, the rest is either a description of what should go next (a command, a directory, an option, etc.) or an actual set of characters to type in. You can always use the **man** command to look up more information for a given command. ***Any command on this list or a command that you would have had to use in order to successfully complete any future lab is a "fair game" question on the mid-term.***

The **man** command: **man**ual pages are on-line manuals which give information about most commands

- Tells you which options a particular command can take
- How each option modifies the behavior of thecommand
- Type **man *command*** at the command line prompt to read the manual fora command

–

- Try to find out about the **mkdir** command by typing: **man mkdir**

- Press the <enter> key to scroll through the manual information line by line
- Press the space bar to scroll through the manual information page by page

- Press the letter q to quit the manual information and return to the LINUX prompt

COMMAND (^ is control)	MEANING
*	match any number of characters
?	match one character
whatis command	brief description of a command
chmod [options] file	change access rights for named file
cd	change to home-directory
cd ~	change to home-directory
cd directory	change to named directory
cd ..	change to parent directory
cp file1 file2	copy file1 and call it file2
wc file	count number of lines/words/characters in file

cat file	**display a file**
tail file	**display the last few lines of a file**
ls -lag	**list access rights for all files**
ls -a	**list all files and directories**
ls	**list files and directories**
who	**list users currently logged in**
mkdir	**make a directory**
mv file1 file2	**move or rename file1 to file2**
man *command*	**read the online manual page for a command**
rmdir directory	**remove a directory**
rm file	**remove a file**
^C	**kill the job running in the foreground**
^Z	**suspend the job running in the foreground**
od	**Dump files in octal and other formats**
jobs	**list current jobs**
kill %1	**kill job number 1**
kill 26152	**kill process number 26152**
ps	**list current processes associated with this terminal**
ps –u kirby.249	**List all current processes for user kirby.249**

Nomenclature:

"." One period, pronounced "dot" means "the current directory I am in regardless of where that might be.

".." Two periods with no spaces between them, pronounced "dot dot" mean "the directory one above the directory I am in."

"/" forward slash is the directory separator in a path.

You can use this to create interesting and useful path expressions. Assume you are in a directory called lab2 and you also have a directory called lab1 and they have the same parent called cse2421. From the lab2 folder you could issue the following command:

cp ../lab1/makefile .

Finally

Some things are easier to retain with repletion. It is strongly suggest that at this time you go back and redo the 2 gdb sections as many times as it takes for you to be able to know what is going to be asked before you get to it. Running gdb, getting the backtrace, reaching into stack frames, printing variables, and setting breakpoints needs to be something you know so well you can teach to others in the class. This is an investment in yourself that will pay off regardless of what editor you use.

For vi users, go and do those tutorials. Make sure that you can:

- Move the cursor while in edit mode, not insert mode
- Use % effectively
- Use > and < (<< and >> also)
- Use the ability to yank text and put it back (vi has rudimentary copy-paste)

There are others to know (change using c comes to mind), but I'll stop there. The big point is *get out of insert mode to edit!*