Lab 2: Makefiles

CPSC 2310 – Spring 2023

CLEMSON School of COMPUTING

Due Date: Monday, January 30, 2023 @ 11:59 PM

The Makefile portion of this lab was originally written by a former TA of mine from several years ago for my class. It has been modified to meet our needs.

Lab Objective

- Learn how to use the Make utility to automate your builds and scripts
- Learn some basic shell scripting
- Write a simple program to read and write data to and from a file.
- Practice with fseek
- Working with multiple files
- Header guards

Part 1: Programing

You will write a simple program that will read data from a file and count the number of sentences and words in the input file. You will print the result to an output file.

Specifications:

I will provide five files that you will use: driver.c, functions.c, and functions.h, input.txt, input2.txt.

You are **required** to use fseek for this lab. You will read through the input file twice, once to count the number of sentences in the input file and once to count the number of words in input file. Hence the use of fseek.

driver.c:

Driver is where main will be implemented. You will use command line arguments to define the names of the files that will be used to read the data from and write the output information to. Main should have minimal amount of code. Ex. It should only be used to define, open, and close file pointers. Check the file pointers opened correctly and call the functions needed to count the number of sentences and words.

functions.h

This file will contain the prototypes of at least 2 functions. You should also add header guards and all needed #include statements. You are welcome to add additional functions if you feel you will need them.

void numSentences(FILE* in, FILE* out);

This function will read the data from the file, counting the number of sentences in the file. For this program you can assume that a sentence will end in either a '.', '?', or '!' . You should also assume that a sentence could have multiple ??, .., or !! at the end.

Your program should print to the output file the following:

Total number of sentences: (the answer)

void numWords(FILE* in, FILE* out);

This function should count the number of words in the input file.

Your program should print to the output file the following:

Total number of words: (the answer)

Example input and output:
First example:

Input:
Hello world!! Hello World..
Hello world. Hello world?? Hello world.
Hello world..
Hello world.

Output:
Total number of sentences: 7
Total number of words: 14

Second example:

Input:
Are you ready to learn?? This is going to be the best semester ever!!! These two sentences should only be counted as sentences once each. So, you cannot just count the number of periods, question marks, or exclamation mark. These sentences are test sentences.

Output:
Total number of sentences: 5
Total number of words: 45

functions.c

You will implement all functions in this file.

FORMATTING:

You will need to add a header to each of your files like the following:

```
/*************************
 *Your name.
 *If you are working with someone both names must be added
 *CPSC 2311 your Section
 *Your email
 *************************/
```

Your program should compile with no warnings and no errors. Points will be deducted for errors and warnings.

- Your code should be well documented. (comments) **Your comments should be in the header files (.h)**
- There should be no lines of code longer than 80 characters.
- You should use proper and consistent indention.

Here are some guidelines for documenting the code in your assignment.
Before each function in the **functions.h** file you are **required** to have a detailed description of what the overall function does. You should explain what each parameter is and what it is used for.

```
/* Parameters: img - image_t pointer array holding the image data for
 *             each of the input files
 * Return:     output - image_t struct containing output image data
 * This function averages every pixels rbg values from each of the
```

Also, if you include comments in the body of the function (and you should) they should be placed above the line of code not beside the code.
Example:

Good

```
//This is a comment
if(something)
{
    do something;
}
```

Bad

```
if(something) //This is a comment
{
    do something;
}
```

Part 2: Makefile
Introduction
In the past, as you've worked on labs and or assignments invariably you've typed a compile command incorrectly or misspelled a filename. As your projects grow, you want to move away from manually typing in the gcc/g++ compile commands and begin using automated build tools.

For Linux-like environments, Make is a very common **build tool** which allows you to compile and link C/C++ programs in a more structured way. Today, we'll begin with simple Makefiles and build to a general-purpose script which will be applicable to many future C/C++ projects.

For those of you that had a lab similar to this, please read the entire lab because it is not exactly like the previous lab.

For this lab you will be required to create a document consisting of screenshots documenting your work. Your screenshots **MUST** include the terminal you are using along with your username. I will do my best to mark the places in the lab that require screenshots. However, this is not guaranteed so be sure to read and follow all directions.

Resources

For a step-by-step tutorial on Makefiles and some information on bash scripting checkout the following links:

https://www.tutorialspoint.com/makefile/

https://www.tutorialspoint.com/unix/unix-what-is-shell.htm

Assignment

Create a folder called Lab2. This is the folder you will use for this part of the lab. Inside Lab2 create another folder called src and move your functions.c and functions.h in this folder.

SCREENSHOT:

Create a document called **Task**. Throughout this lab you will be instructed to take screenshots; adding them to this document.

What you should have now is:

A folder called Lab2 – In this folder you should have another folder called src.

/Lab2/src

Lab2 should have your driver.c, and a folder **src** which should have funtions.c, functions.h and the Task document.

Because of the above actions you are going to need to change your driver.c file. Determine what that change is and make it.

In your Task file describe what the change was and why you needed it.

As projects get larger and include more source files, it often makes sense to logically separate them into subdirectories to make them easier to manage. Even if you don't split them up, compiling multiple source files into an executable becomes more involved and error prone process. Especially when multiple files have compiler errors.

This is where the make command is very useful. Make allows us to define a script called the makefile which contains instructions on compiling our programs. In the root directory (Lab2) of the code listed above, create a file named makefile by doing the command below:

```
touch makefile
```

Touch is a Linux/Unix command that is used to create an empty file with no content.

SCREENSHOT

Make a screenshot of this and copy/paste it in your Task document.

Note that the file MUST be named **Makefile** or **makefile** for the make command to identify it.

Makefile Targets and Dependencies

Makefiles are a form of **bash script**, which is like a programming language for controlling your Linux terminal. It allows us to compile source files and run additional programs like **echo**, **grep**, or **tar**. Consider the following:

```
MESSAGE=HELLO WORLD!

all: build
   @echo "All Done"
build:
  @echo "${MESSAGE}"
```

The first thing to notice is that we defined a **bash variable** at the top of the script. Bash variables are simply **string substitutions**. Using the **$() or ${}** operator, we can replace the variable with the defined string. Use **${}** with curly braces if you are in between double quotes. This is a pretty straightforward concept, but it's very powerful. In this instance, we use a variable to define a message we print to the screen.

****

HUGELY IMPORTANT NOTE:

Any code that belongs to a given target must be indented at least one TAB character. There is a tab before both of the @echo commands above.

****

SCREENSHOT:

Using the file you created with the command touch, type the above code in the makefile. Create a screenshot and paste it in your document

Now we move on to our first **target,** the default target **all:**

A target is simply a named chunk of code our Makefile will try and execute. By default, Make will execute the **all:** target if it exists. Otherwise, it executes the **first target it finds**. You can also specify a target from the command line, which means that the following commands are **equivalent**:

```
make
make all
```

Next to the target name there is a whitespace separated list of **dependencies**. When make encounters a target to execute, it will read the dependency list and perform the following actions:

1. If the dependency is another target, **attempt to execute that target**
2. If the dependency is a file that matches a rule, **execute that rule**

This means we can **chain multiple targets together**.

Run the make command from the terminal and see what is printed to the screen:

```
make
HELLO WORLD!
All Done!
```

Notice that the "HELLO WORLD!" is printed first… that's because the all: target is **dependent** on the build: target and so it is run first.

Now run the following

```
make build
```

SCREENSHOT:

Take a screenshot showing that you ran the make command and make build. Paste the screenshot in your **Task** document.

Add a target called tar: that will tar zip everything in the folder, your tar file should be test.tar.gz.

Add a target called untar: that will untar the tared test.tar.gz file.

Now add a target called list: that will list all files in the folder. In other words, if I type make list it should list all files in the folder. If I type make tar it will tar all files in the folder creating a file named test.tar.gz. Test your make tar and make list.

SCREENSHOT:

Create a screen shot and copy/paste it in your document.

Now, change the **all** target to have dependencies of build, tar, and list.

SCREENSHOT:

Test it, create a screenshot, and place it in your document.

For those wondering, **echo** is a command that just prints its arguments to the terminal. Including the @ symbol in front of the command **prevents make from printing the command**. Play around with the above file to get a feel for what we are doing. See what happens if you remove the @ symbol.

Making C Programs

The power of the makefile is in this dependency list/rule execution loop. What we want is for our build target to **run our gcc compile command**. This target should be **dependent on our C source files**. A powerful feature of make dependencies is that **a target will only be executed if its dependencies have changed since the last time you called make**.

Again:

A target will only be executed if its dependencies have changed since the last time you called make!

Let's just skip to an example!

Using the command **mv**, change the name of the previous makefile you created. Use touch and create another makefile. Type the following code in the makefile filling in the C_SRCS and HDRS information.

```
# Config, just variables
CC=gcc
CFLAGS=-Wall -g
LFLAGS=-lm
TARGET=out

# Generate source and object lists, also just string variables
C_SRCS := FILLIN THE C FILES HERE
HDRS := FILL IN THE HEADER FILES
OBJS := driver.o src/functions.o
```

```
# default target
all: build
                @echo “All Done!”

# Link all built objects
build: $(OBJS)
                $(CC) $(OBJS) -o $(TARGET) $(LFLAGS)

# special build rule
%.o: %.c $(HDRS)
                $(CC) $(CFLAGS) -c $< -o $@
```

SCREENSHOT:

Run your make file, your output should look something like the following. Take a screenshot and put it in your document.

```
make
gcc -Wall -g -c driver.c -o driver.o
gcc -Wall -g -c src/functions.c -o src/functions.o
gcc driver.o src/functions.o -o out -lm
All Done
```

Take a deep breath… this is a lot so let’s go over it! First, we **define some new variables** using an alternative **assignment operator** “:=”. Nothing special here, just an assignment and string substitution. The C_SRCS and HDRS variables should make sense, but this might be the first time you’ve seen the **.o files** defined in the OBJS variable.

A “.o” file is called an **object** file, and is basically one step away from machine runnable code. Open one in a text editor and look at it yourself. We use the **-c gcc flag** to generate these files, and they are useful as an intermediate step before producing our executable. Using this in between step, it is possible to **compile source files individually** to isolate compiler errors before linking.

Note that our **build target** depends on these object files. This means that make **will look for these files in our directory, and try to create them if they are missing.**

To create these object files, we define a rule or recipe:

```
%.o: %.c $(HDRS)
                $(CC) $(CFLAGS) -c $< -o $@
```

This **rule** applies to **any file that ends in .o**. So, when build tries to resolve the $(OBJS) dependencies, it will run the rule once for every file in our list. The **%** sign is a **wildcard** in this case, replaced by the path to the file we are processing.

Note that this rule to make an object file **also has dependencies**. In this case, the object file depends on the source file of the **same name** and all the headers in the program. This means that **if the C source or any header is changed between makes, we will then recompile the object.** The compile command is straightforward except for the $< and $@ variables:

- $< evaluates to the first dependency (so, %.c)
- $@ evaluate to the name of the rule (so, %.o)

This allows us to write **one rule that covers all object files in our OBJS list**. For every object file we need, we will run an individual compile command. Note that this rule will run if the .o file **does not exist** or if the **dependencies have changed** since last make. Basically, Make checks the "last altered" time of the C source file and the object file to see if they are different.

It might not be easy to see, but all this means is that as we work on larger projects and need to recompile, **only object files which need to be updated will be recompiled.** This saves us time and reduces the number of unneeded recompiles.

Once all the object files are up to date and created, we go back to the build target and link them all together to produce our executable TARGET.

It's not a big deal if this doesn't make complete sense right now, but play around with the makefile above and it should come together. At the end of the day, it just has to work!

Advanced Makefile Commands and Rules

All this is great, but really, we just moved typing out file lists from the terminal to a file. What would be useful is if the Makefile could **find our source files by itself**. Linux has many useful functions to help manipulate files in our project, and we can access these tools from our Makefile. You can include conditionals as well as filters to really customize your build process. For our purposes, let's just use the simple **wildcard** function:

TASK

Type the following.

```
C_SRCS := \
  $(wildcard *.c) \
  $(wildcard src/*.c) \
  $(wildcard src/**/*.c)

HDRS := \
  $(wildcard *.h) \
  $(wildcard src/*.h) \
  $(wildcard src/**/*.h)
```

Here we are executing the **wildcard** command to match all files with the .c and .h endings. This includes all files in the "src" folder and all its subdirectories. The \ is used to break a command over more than 1 line.

Let's add a new target called "which" to see what these wildcard commands do:

```
which:
		@echo "FOUND SOURCES: ${C_SRCS}"
		@echo "FOUND HEADERS: ${HDRS}"
```

Run "make which".

SCREENSHOT:

Make a screenshot of the output and paste it in your documents.

Read through the explanation of the remaining task. At the end the completed make file is given.

Now that we've successfully collected our headers and source, we need to take the list of sources and generate an appropriate list of object files. Things are about to get a little tricky:

```
OBJS := $(patsubst %.c, bin/%.o, $(wildcard *.c))
OBJS += $(filter %.o, $(patsubst src/%.c, bin/%.o, $(C_SRCS)))
```

Don't panic. The first line uses the **patsubst** command to generate a list of strings where we replace the .c ending with a .o ending. **patsubst** stands for **pattern substring** and replaces a pattern found in source files (such as the extension!). In this case, we use the wildcard function again so that **we only process files in the root directory**. Note another tweak: we've also tacked on a "bin/" in the front of all these object files when we did the substitution. More on that later.

Next, we get the list of object files needed from the src directory and its children. Here we are using the **filter** function to exclude any results which don't end in ".o". This is necessary if we have source files in both the root directory and source directory because root directory files get processed twice.

Don't worry too much if this step is confusing, just know that it allows us to collect all our source, header, and object files with only 4 lines of code!

The last step is to tweak our **rules** used in compiling object files. Remember that "bin/" we added to the beginning of all our objects? We did this so that the compiled objects are **saved to their own bin/ directory** instead of cluttering up our src folder.

```
# Catch root directory src files
bin/%.o: %.c $(HDRS)
		@mkdir -p $(dir $@)
		$(CC) $(CFLAGS) -c $< -o $@

# Catch all nested directory files
bin/%.o: src/%.c $(HDRS)
		@mkdir -p $(dir $@)
		$(CC) $(CFLAGS) -c $< -o $@
```

We **simply add a bin/ to the front of the rule** to match our new object file names. We also create a duplicate rule which looks for **src/%.c** files instead of simply %.c files. This is needed because we are no longer saving our .o files next to our source files, and the paths don't match by default.

There's also a call to **mkdir** in these rules, which will create the bin/ directory and its subfolders if necessary. Here we use the **dir** command to get the directory prefix of our object file (remember the file name is the same as the rule name $@).

Now if you run "make" you will be able to compile all your source files into objects and store those object in the bin/ directory **which will be created by make**. Since our original **build** target depends on the OBJS list, it will automagically know to use this bin/ directory to link your program together.

The only thing we have left to do is to add a **clean** target and a **run** target to allow for fresh builds and quick execution respectively. Add these to the bottom of your Makefile:

```
clean:
                rm -f $(TARGET)
                rm -rf bin


run: build
                ./$(TARGET) <command line arguments needed go here>
```

Again, run these targets using "make clean" or "make run". Notice that run depends on our build target, so the program will be **compiled if needed before we try and run the executable**.

Test out the makefile and run the included code if you wish. This Makefile is generic and will work on most C projects. For very large projects with modules and outside libraries, it is possible to generate dependency lists, include other makefiles, have conditional rules, and all manner of other witchcraft.

Make is powerful, if you code in C or C++ it is one of the most important tools to learn. Other compiled languages like Java or TypeScript have similar dependency tools like **Maven**, **Gradle**, and **yarn**. Out in the industry they are must haves for efficient development, so keep a lookout and be ready to learn new tools to make your programming life easier!

Tips and Tricks

Here is the completed Makefile described above. You are to carefully type this makefile and add detailed comments. These comments should be long enough to convince your TA that you know and understand what each section of this makefile is doing. Most of the explanation is described in this document. You are not allowed to simply copy and past. Paraphrase if you wish, but make sure the grader is convinced you know what this make file is doing.

```
# add comments
CC=gcc
CFLAGS=-Wall -g
LFLAGS=-lm
TARGET=out


# add comments
C_SRCS := \
  $(wildcard *.c) \
```

```
  $(wildcard src/*.c) \
  $(wildcard src/**/*.c)
# add comments
HDRS := \
  $(wildcard *.h) \
  $(wildcard src/*.h) \
   $(wildcard src/**/*.h)
# add comments
OBJS := $(patsubst %.c, bin/%.o, $(wildcard *.c))
OBJS += $(filter %.o, $(patsubst src/%.c, bin/%.o, $(C_SRCS)))

# add comments
all: build
                    @echo "All Done"

# add comments
build: $(OBJS)
                    $(CC) $(OBJS) -o $(TARGET) $(LFLAGS)

# add comments
bin/%.o: %.c $(HDRS)
                    @mkdir -p $(dir $@)
                    $(CC) $(CFLAGS) -c $< -o $@

# add comments
bin/%.o: src/%.c $(HDRS)
                    @mkdir -p $(dir $@)
                    $(CC) $(CFLAGS) -c $< -o $@



#add comments
clean:
                    rm -f $(TARGET)
                    rm -rf bin
#add comments
run: build
                    ./$(TARGET) < put your command line arguments here>
#add comments
which:
                    @echo "FOUND SOURCES: ${C_SRCS}"
                    @echo "FOUND HEADERS: ${HDRS}"
                    @echo "TARGET OBJS: ${OBJS}"
```

Compile and Execute

Use Make to compile your code as follows:

```
make
```

Execute the program

```
make run
```

You should now know enough about makefiles to create your own simple makefile.

The information in this lab only scratches the surface of what the make utility can actually do. I encourage you to explore the make utility.

Submission Instructions

Use handin.cs.clemson.edu to submit your files.

Things to do prior to handing in your files.

1. **Test your program on the SoC servers**. To ensure everyone is using the same version of Linux, you are required to test your program on the SOC's **cerf15.** There are many ways to access cerf15. If you are unsure of how, ask your lab TA. This is the machine your program and makefile will be tested on.
2. **FILL IN WITH SPECIFIC INSTRUSTIONS FOR THE MAKEFILE TO MAKE IT EASIER TO GRADE.**
3. Tar zip all of your lab 2 files naming the tarred file Lab2<YourUserName>.tar.gz.

How to get a 100 on the autograder

1. Take the screenshots
2. Submit only a .tar.gz file
3. Provide a working makefile that will pass and behave as expected on ‘make run’
4. Provide all targets from final makefile
5. Have a working clean that removes the bin folder and the executable
6. Do not include a ton of std lib headers in main
7. Provide header guards
8. provide your username and email in each file (without the g. please)
9. Use fseek and FILE ptrs
10. Pass all ten test cases.

Full warning I wrote the autograder very quickly. If you find an issue please simply provide a short explanation either to me in person or via email at jhembre@g.clemson.edu. This is Judsen Hembree typing btw.