Lab 3 – Fun with Structs
CPSC 1021 – Fall 2019

CLEMSON
School of COMPUTING

Due Date: 20 September, 2019 @ Midnight

Lab Objective

- Practice building multiple file programs
- Build comfort with pointers of structs
- Practice checking for bad user input
- Build complex makefiles
- Continue to build comfort with pointers and dynamic memory allocation.

Introduction

As you've been working on lab invariably you've typed a command in incorrectly or misspelled a filename. As your projects grow, we 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 projects.

This lab consist of 2 parts; 1. build familiarity with makefiles in C; 2) work with structures. Understanding the core idea of structures will help your transition to Object Oriented Programming (OOP) when we move to C++. Todays lab will use an array of structures in the form of a pointer. Your program will be a basic collection of students with various information collected about the student.

You must implement the given functions in the .h file.

Part 1

Resources

For a step by step tutorial on Makefiles, check the link here:

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

Also, some information of bash scripting

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

As projects get larger and include more source files, it often makes sense to logically separate them into sub directories 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 (Lab3) of the code listed above, create a file named **Makefile** by doing the command below:

```
touch makefile
```

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.

****

YOU SHOULD TYPE THE ABOVE IN THE MAKEFILE YOU CREATED WITH TOUCH

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 white-space 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.

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!

```
# Config, just variables
CC=gcc
CFLAGS=-Wall -g
LFLAGS=-lm
TARGET=lab3

# Generate source and object lists, also just string variables
C_SRCS := src/main.c src/functions.c
HDRS := src/functions.h
OBJS := src/main.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 $@
```

The output of running this makefile looks like this:

```
make
gcc -Wall -g -c lab3.c -o lab3.o
gcc -Wall -g -c src/main.c src/functions.c -o src/main.o src/functions.o
gcc lab5.o src/main.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: %.cpp $(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:

```
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}"
```

You can run this target with "make which" and it produces output:

```
make which
FOUND SOURCES: lab3.c src/main.c src/functions.c
FOUND HEADERS: src/functions.h
```

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.

Erase the old object file rule and use these two rules to correctly compile our objects:

```
# 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)
```

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 ppm 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 you wrote today:

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

# Generate source and object lists
C_SRCS := \
  $(wildcard *.c) \
  $(wildcard src/*.c) \
  $(wildcard src/**/*.c)

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

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

all: build
                    @echo "All Done"

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

# 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 $@




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

run: build
                    ./$(TARGET)

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
```

Part 2:
I recommend you read through the entire instructions for Part 2 to ensure you understand the requirements for this assignment. I strongly recommend you spend time planning your algorithms prior to coding.

I have provided you with **functions.h**. This file defines the struct called Student. It also has 3 functions you are responsible for implementing. There is 1 additional function that is extra credit. Keep reading to find out which is extra credit. Below I will give you a brief overview of each function. **You are NOT allowed to change the signature of any of the function prototypes.** A signature of a function consists of; 1. the return type, 2. the name of the function, and 3. the number and type of parameters for the function.

void addStudents(struct Student* s, int *numStudents);
Parameters:
struct Student* s: this is an array of type struct Student that will hold student information
int *numStudents: this is a pointer to a variable that keep track of the number of students that have been added to an array of Students.

Steps for this function:

- You will ask the user to enter the information of the student they would like to add.
- For each piece of information, read the data and store it in the appropriate members of the struct.
- NOTE: Although you have allocated the memory for the array of Students, you still need to dynamically allocate the memory for the char* firstName. You can assume firstName will not be more than 25 characters.
- Use the function **strcpy** to copy the student name in the char

If you do not know how to use strcpy, you should look up the documentation for this function.

void printStudents(struct Student* s, int *numStudents);
Parameters:
struct Student* s: this is an array of type struct Student that will hold student information
int *numStudents: this is a pointer to a variable that holds the number of students added to the array of Students.

Steps for this function:

- Loop through each element of the array of students printing the information in the following format:

Student #1
Student's Name: Damion
Age: 25
CUID: 1234
GPA: 3.45

Student #2
Student's Name: Yvon
Age: 55
CUID: 1235
GPA: 3.67

int printMenu();

This function prints the following message and returns the value entered by the user.

Please select an option:
1. add a student
2. print students currently saved
3. end program

NOTE: IF YOU ARE DOING THE EXTRA CREDIT LISTED BELOW YOUR MENU OUTPUT SHOULD BE AS FOLLOWS:

Please select an option:
1. add a student
2. print students currently saved
3. change GPA of a Student
4. end program

EXTRA CREDIT (5 points):
If you choose to implement this function you will receive 5 points extra credit.

void changeGPA(struct Student* s)
Parameter:
struct Student* s: this is an array of type struct Student that will hold the student information

Steps for this function:

- Ask the user to enter the CUID of the student you want to change the GPA.
- Loop through the array of students to find the student that has that CUID.
- If the student is found confirm it indeed is the student you are looking for by showing the user the name of the student associated with the CUID they entered.
- If this not the student they are looking for, ask the user to enter the correct CUID.
- If this is the correct student ask the user to enter the new GPA.
- If the given student CUID is not found, inform the user the student could not be found.

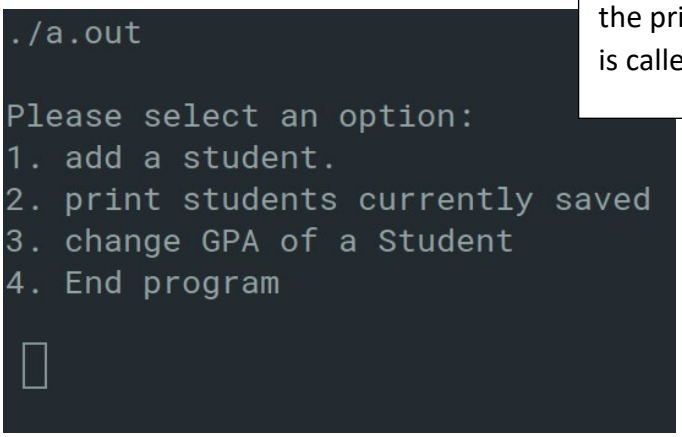
NOTE: As a general rule myself nor the TA will provide help with Extra Credit. You are on your own w.r.t. EC.

Main.c

This is the driver file that holds **int main();**
In the main function you will need to dynamically allocate memory for an array of 10 Students. To start the process you need to call the printMenu() function. Based on the return value from printMenu() you need to call one of the functions from the functions.h file.
addStudents, printStudents, printMenu, or changeGPA

A suggestion is that you create a Boolean variable. Use the Boolean in a while loop to continue to loop until the user enters a 4. Option 4 in the menu is End Program. This should set the Boolean to false which should stop the while loop. Lastly, do not forget to release the dynamically allocated memory. You have to release the memory for the 10 students. However, before you do this you should release the memory set aside for each students firstName.

Sample output when the printMenu function is called



Sample output when printMenu is called; 1 is chosen. Student info was entered, then the menu is displayed again. Next option 2 is chosen and the current student's information is displayed and the menu is displayed again. This continues until "end program" is chosen

```
./a.out

Please select an option:
1. add a student.
2. print students currently saved
3. change GPA of a Student
4. End program

 1
what is the name of the student you'd like to add?  Damion

How old is the student?   25

What is his / her CUID number?    1234

What is the students GPA? 3.45

Please select an option:
1. add a student.
2. print students currently saved
3. change GPA of a Student
4. End program

2

Student #1
        Student's Name: Damion
        Age: 25
        CUID: 1234
        GPA: 3.45


Please select an option:
1. add a student.
2. print students currently saved
3. change GPA of a Student
4. End program
```

```
Please select an option:
1. add a student.
2. print students currently saved
3. change GPA of a Student
4. End program

3
What is the student's CUID?     7658
Sorry, that CUID was not found

Please select an option:
1. add a student.
2. print students currently saved
3. change GPA of a Student
4. End program

3
What is the student's CUID?     1234
CONFIRM: Students name is damion
1 for yes, 0 for no     1

Insert new GPA: 4.0

Please select an option:
1. add a student.
2. print students currently saved
3. change GPA of a Student
4. End program

2

Student #1
        Student's Name: damion
        Age: 25
        CUID: 1234
        GPA: 4.00
Student #2
        Student's Name: jeremy
        Age: 24
        CUID: 5432
        GPA: 3.50
```

This is a sample when “change of GPA of a Student” is chosen. Remember this is an EC function. Make sure you read the instructions for this function. You are not required to complete this function.

The CUID is requested. The CUID is not found and a message is printed followed by the menu again. A valid CUID is entered, the student info is displayed and user is asked to confirm. If confirmed the user is prompted for the correct CUID, then the menu is displayed again. The function printStudent is called and the students in the array are printed.

What to hand in

Once you have confirmed your program works, you have commented your code, and it satisfies the requirements outlined in this document, you must hand in the following files:

- main.c
- functions.c
- functions.h
- makefile

FORMATTING

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

```
/*************************

 *Your name

 *CPSC 1021, your Lab Section, F19

 *Your email

 *Course Instructor

 *************************/
```

Your program should compile with no warnings and no errors. If you program will not compile you will receive a **0** on the lab.

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

Any infractions of the above listed requirements will result in a 5 point deduction for each infraction category.

All .h files should include header guards.

Turning in Your Assignment

Handin

You will use the School of Computing ***handin*** system to submit your lab document. The document should be in pdf format and named <username>_lab3.pdf.

Most labs will require you to use the ***tar*** utility. The term "tar" stands for tape archive, an archiving file format. You will use the ***tar*** utility to "tar zip" (tar.gz) all files prior to submitting through ***handin***. When you have completed the entire lab, save your document as a pdf. Make sure you are inside the current lab1 directory and your pdf and C files are inside the same directory. Using the terminal, ***tar*** your file as follows:

```
tar –czvf <username>_lab3.tar.gz *
```

The <username> is your username.

Let's break this down:
tar – the tar utility command
-czvf are flags used by the tar utility

c – create a new archive containing the speciied items
z – (c mode only) Compress the resulting archive with ***gzip***
v – verbose output (tar will list each file name as it is read from or written to the archive)
f – read the archive from or write the archive to the specified file
username_lab3.tar.gz - this is the file name. This must come before the list of names of the files to ***tar***.
*- this is the wildcard. This will archive all files in the current directory. If you do not use the wildcard, you must type the name of each file you want archived.

Once you have tarred your file you should always check the files in your tarred file to determine if they are all there and in workable order. The command to ***untar*** a file is as follow.

```
tar –zxvf <username>_lab3.tar.gz
```

Untar your tarred file and let your TA see that you have indeed untarred the tarred file and that the archive file was not corrupt.