Lab 7: The Making of the Makefiles

CPSC 1021 – Spring 2020

CLEMSON School of COMPUTING

Due Date: Friday, 2/28/202 before 5pm EST.

Introduction

During this lab you will:

- Learn how to use the *make* utility to automate your builds.

Part I

As you are working at the projects that consist of multiple files, you have noticed that you need to type several commands to compile all you files to object code and then link them together. Is there a better/faster way to do it? Yes, there is! There is also a way to automate many tasks by writing small bash scripts.

For Linux-like environments, make is a very common build tool which allows us to compile and link C/C++ programs in a more structured way. Let us look at this utility and the makefiles that it uses to build your programs.

1. For a step by step tutorial on makefiles, please click on this link:

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

   Go through the tutorial on the left hand side. Click through the links and read the information. Some of you are already familiar with basic makefiles, but this is still a good review. Also, you can search online for examples of makefiles.

   Also, some information of bash scripting that may be helpful to understand some of the syntax: https://www.tutorialspoint.com/unix/unix-what-is-shell.htm

2. Please note that the name of the utility (program) itself is “make”. It uses a script file saved as a Makefile, or makefile to build your program. The name of the file can start either with an upper case, or a lower case letter, and it has no extensions. The file contains a script that instructs compiler how to compile your program. To run the utility, you will type make on the command line. The utility will then search your current directory for a file named Makefile (with uppercase M), and if it does not find it, it will then look for a makefile, in that order. It will then execute that file. Let’s look at some makefile components.

   a. Targets and Dependencies

Makefiles are written in a bash script, a scripting language used in the Linux bash shell. A shell is a programming environment (your terminal window, in this case) that allows you to execute commands on the command line. It allows us to compile source files and run other utilities/programs, such as echo, grep, or tar. There are several different shells that use different scripting languages, but Linux uses bash shell by default. The name “bash” stands for “bourne again” shell, which is a spin off the older “bourne shell”. So, consider the following:

```
MESSAGE=HELLO WORLD!
all: build
     @echo "All Done"
build:
     @echo "${MESSAGE}"
```

On top we defined a bash variable called MESSAGE. Bash variables are simply string substitutions. Using the $() or ${} operator, we can replace the variable with the defined string literal. Use ${} with curly braces, if you use double quotes. This is a pretty straightforward concept, but it’s very powerful. In this case, we used a variable to define a message we print to the screen.

Important: *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.*

The first target is the default target all:.A target is simply a named piece of code our makefile will try to execute. By default, make will execute the all: target first, 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 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. If we 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.

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

b. 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 g++ 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.

Important: *A target will only be executed if its dependencies have changed since the last time you called make! Here is another example!*

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

# Generate source and object lists, and string vars
C_SRCS := lab2.c src/ppm/ppm_utils.c
HDRS := src/ppm/ppm_utils.h
OBJS := lab2.o src/ppm/ppm_utils.o

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

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

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

The output of this makefile looks like this:

```
make
gcc -Wall -g -c lab2.c -o lab2.o
gcc -Wall -g -c src/ppm/ppm_utils.c -o src/ppm/ppm_utils.o
gcc lab2.o src/ppm/ppm_utils.o -o out -lm
All Done
```

So, what is going in this script? We have defined 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 file with an extension ".o" is called an object file. For curiosity's sake you can 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:

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

This rule applies to any file that ends in .o. 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.

c. 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: lab2.c src/ppm/ppm_utils.c
FOUND HEADERS: src/ppm/ppm_utils.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) puppy.ppm out.ppm
```

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 etc.

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!

Here is the completed makefile you wrote today:

```
# Config
CC=gcc
CFLAGS=-Wall -gLFLAGS=-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) puppy.ppm out.ppm
which:
          @echo "FOUND SOURCES: ${C_SRCS}"
          @echo "FOUND HEADERS: ${HDRS}"
          @echo "TARGET OBJS: ${OBJS}"
```

To compile and execute your makefile you will type : `make` and then: `make run`

Part II.

1. Last week you have written a program that consists of 5 files: owner.cpp, owner.h, dog.h, dog.cpp and main.cpp (your names may be different). We will use these files here to write a makefile, so please keep those files handy. Examples shown above use C programs as example, but will be similar to build C++ programs.

2. Create a makefile using the example makefile shown above. Test and make sure everything works.

What/How to submit

Create a tarball consisting of your 5 program files and the new makefile and submit your zipped tarball to canvas to be graded.

NOTE: As before, if you submit the archive that cannot be open, or is corrupt, you will not be given the opportunity to redo the work and resubmit. You have one shot to get it right.