Lab 06 - Makefiles

Due: October 29, 2017 at 4:00pm

Due at lab hours - NOT Autograded

1 Introduction

In this lab, you will learn how to write your own makefiles. Up to this point in the course, we have supplied you with the makefiles needed to build your projects. After this lab, you'll have the tools to write your own makefiles for your projects!

To get started, install the lab stencil by running:

```
cs0330_install lab06
```

2 Makefiles

2.1 Intro

A makefile is essentially a script that simplifies compilation, cleanup, and other tedious or repetitive tasks necessary to build a project. In this class, we use makefiles primarily to compile C code, but they can be used to simplify any set of commands.

You can execute instructions in a makefile by running **`make`** in the same directory as the makefile. If you have several makefiles, and one of them is named **`MyMakefile`**, then you can execute it with the command: **`make -f MyMakefile`**. If no **`-f`** option is present, make will look for the makefiles **`GNUmakefile`**, **`makefile`**, and **`Makefile`** in that order. For more info on the **`make`** utility, type **`man make`** in a department machine.

2.2 Writing a Makefile

2.2.1 Basic Makefile and Targets

To compile by hand in terminal, you would usually type

```
gcc -Wall -Wunused -Wextra -std=c99 life.c -o life
```

Typing this every time you want to recompile can be tedious. As projects get larger and involve more files, makefiles become very helpful because they allow you to use a single command to build an entire project. The basic *structure* of a makefile is as follows:

```
target: dependencies
[tab] shell command
```

A *target* is a label that denotes a specific task or set of commands to run. Multiple shell commands can be run for a single target by placing each shell command on a new line following the target, making sure that each line begins with a tab. The target and set of commands is sometimes referred to as a *rule*. A *dependency* (sometimes called a prerequisite) is either a filename or the name of another target upon which this target depends.

To run a specific target, you run the command **`make <target name>`** in a shell.

To create a very basic Makefile for our **`gcc`** command above for the life lab, you would create a file named **`Makefile`** and write

```
life:
    gcc -Wall -Wunused -Wextra -std=c99 life.c -o life
```

In this first example we see that our target is called **`life`**, which is also the name of the executable that's produced by this rule. This is no coincidence! The shell commands included for a target typically result in the creation of a single file. If this is the case, the name of the target should match the name of this new file. Behind the scenes, **`make`** will check if any of the dependencies for a target were modified more recently than the target file itself. If so, the shell commands for that target are run. Otherwise, **`make`** will not run the shell commands as the target should be up to date with its dependencies.

2.2.2 Using Dependencies

As mentioned in the previous section, *dependency* can be a filename or the name of another target upon which this target depends. If a dependency is a filename, **`make`** will only execute the target's commands if the file has changed since the last **`make`**. If it is a target name, **`make`** will run the dependency target first and then run the commands in this target. Any target can have multiple dependencies. Furthermore, if a dependency file or target does not exist, then **make** will raise an error.

Our makefile from above, with dependencies:

```
life: life.c life.h
    gcc -Wall -Wunused -Wextra -std=c99 life.c -o life
```

2.2.3 Multiple Targets

It's often useful to have more than one target. These different targets can build different parts of your project, or build it in different ways, or do something else entirely. But, if you don't specify a target, **make** will by default run only the first rule defined in the makefile. If you want to build more than one rule by default, you'll need your first rule to be some target that *depends on* the other targets. The name of this "super-target" is, by convention, **all**. For example:

```
all: life hello

life: life.c life.h
    gcc -Wall -Wunused -Wextra -std=c99 life.c -o life

hello: hello.c
    gcc -Wall -Wunused -Wextra -std=c99 hello.c -o hello
```

It's also customary to write a **clean** target that removes all build output:

```
clean:
    rm -f life hello
```

2.2.4 Phony Targets

You may have noticed that it's possible for a target to not correspond to the name of any build output; in the example above, these are **all** and **clean**. These targets are known as "phony" targets. However, if we don't explicitly label phony targets as such, **make** will look for files with the target name, as if they were normal targets. So, if we create files named **all** or **clean** in the same directory as the makefile, those rules won't run properly.

We can explicitly label phony targets by including the line **.PHONY: <target1> <target2>...** somewhere in the makefile. For example:

```
.PHONY: all clean

all: life hello
```

```
life: life.c
	gcc -Wall -Wunused -Wextra -std=c99 life.c -o life

hello: hello.c
	gcc -Wall -Wunused -Wextra -std=c99 hello.c -o hello

clean:
	rm -f life hello
```

Note that makefiles can do more than just compiling your projects. You can have targets in the makefile that compile a LATEX file into a pdf, execute tests, or run any other shell commands (as with the **`clean`** rule above).

A trivial example:

```
printCS33Banner:
	banner hello CS033!
```

NOTE: You *must* use tabs for indentation in your Makefiles. In other words, if you are using a text editor such as vim and choose to record your tabs as spaces, then the Makefile will not work correctly.

2.2.5 Variables and Comments

You may have also noticed that the rules above were somewhat repetitive. We had several rules that invoked gcc with many of the same flags. If we wanted to add, change, or remove a flag, we'd normally have to do so for each rule, which is the exact kind of tedium that makefiles are supposed to eliminate in the first place! Fortunately, makefiles support variables (and also comments):

```
#This is a comment.
#CC, CFLAGS and EXECS are variables.
CC = gcc
CFLAGS = -Wall -Wunused -Wextra -std=c99
EXECS = life hello

.PHONY: all clean

all: $(EXECS)

life: life.c
	$(CC) $(CFLAGS) life.c -o life
```

```
hello: hello.c
    $(CC) $(CFLAGS) hello.c -o hello

clean:
    rm -f $(EXECS)
```

Note that while using variables appropriately is not a requirement to be checked off for this lab, future assignments will expect that any makefiles you write are well-formatted and succinct and will deduct points for any failures to do so. As such, it is *highly recommended* that you use variables in this lab for the sake of receiving full makefile points on future assignments.

2.2.6 Automatic Variables

Finally, we will briefly discuss automatic variables. These are variables that are defined by each target rule. Some helpful automatic variables are listed below. For a full list, view **`make`**'s documentation.

- **`$@`** : The name of the target.
- **`$<`** : The name of the first dependency.
- **`$^`** : The names of all the dependencies, with spaces between them.

Example:

```
#This is a comment.
CC = gcc
CFLAGS= -Wall -Wunused -Wextra -std=c99
EXECS = life hello

.PHONY: all clean

all: $(EXECS)

life: life.c
    $(CC) $(CFLAGS) $< -o $@

hello: hello.c world.c
    $(CC) $(CFLAGS) $^ -o $@

clean:
    rm -f $(EXECS)
```

3 Assignment

Monsters, Inc. needs your help! Mike Wazowski and Sully have been spreading rumors that scaring humans is immoral. As the brand ambassador at Monsters Incorporated, you must convince Monstropolis that Monsters Inc is a legitimate business, using the catchy slogan “we scare because we care.” It is your job to write a Makefile to deliver your message to the citizens of Monstropolis in two volumes: LOUD (for the monsters with smaller ears) and quiet (for the monsters with larger ears).

Here is an example of how you intend to use the quiet and loud REPLs (Read Eval Print Loop). Each REPL reads lines of user input and then alternates shouting or whispering them back to the user. When a line is whispered, all letters are converted to lowercase. When a line is shouted in the quiet REPL, the first letter of each word is capitalized. When a line is shouted in the loud REPL, all letters are capitalized. The following is an example of using both the quiet and loud REPLs:

```
$ ./quiet_repl
We scare because we care
Shout: We Scare Because We Care

We scare because we care
Whisper: we scare because we care

$ ./loud_repl
We scare because we care
Shout: WE SCARE BECAUSE WE CARE

We scare because we care
Whisper: we scare because we care
```

3.1 Description

This lab contains the following files:

- **`upper.c`**: a C file defining our **`to_uppercase()`** function.
- **`upper.h`**: a header file declaring our **`to_uppercase()`** function.
- **`repl.c`**: a C file that creates a REPL that alternates whispering and shouting lines of user input back to the command line. If the **`-DEXTRA_LOUD`** compiler flag is included, lines will be shouted back in all caps, instead of having the first letter of each word capitalized.
- **`repltests`**: a script which runs tests on both REPLs you will create. This script is used with the following syntax: **`./repltests <quiet_repl> <loud_repl>`**.
- **`Makefile`**: a Makefile where you will define rules for the targets listed below.

- **`README.tex`** : a LATEX README file that can be compiled into a beautiful README pdf.

In addition, the following files are located in the **`/course/cs0330/pub/labs/makefile_lab`** directory.

• **`lower.c`**: a C file defining our **`to_lowercase()`** function.
• **`lower.h`**: a header file declaring our **`to_lowercase()`** function.

In this lab, you will write a Makefile that will build two different REPLs from the provided header and c files. If you examine **`repl.c`** you will notice that it includes both **`upper.h`** and **`lower.h`**, but **`lower.h`** is not initially in your directory. As part of your Makefile, you will need to copy over **`lower.c`** and **`lower.h`** into your local directory.

Specifically, your Makefile must define rules for at least the following targets:

- **`quiet_repl`**: This should create the **`quiet_repl`** executable, which shouts lines back to the user with the first letter of each word capitalized.
- **`loud_repl`**: This should create the **`loud_repl`** executable, which shouts lines back to the user with all letters capitalized.
- **`README.pdf`**: This should create a pdf from the provided **`README.tex`**. A LATEX file can be compiled into a pdf with the command **`pdflatex <input.tex> <output.pdf>`** where **`input.tex`** in the input LATEX file and **`output.pdf`** is the desired name of the newly created pdf file.
- **`all`**: This should create **`quiet_repl`**, **`loud_repl`**, and **`README.pdf`** in the current directory.
- **`clean`**: This should remove all the files that were made by the Makefile. Be sure to remove both **`lower.c`** and **`lower.h`** as well.
- **`test`**: This should use the provided **`repltests`** script to run tests on both REPLs.

Feel free to write other rules for commands shared between targets.

3.2 Testing

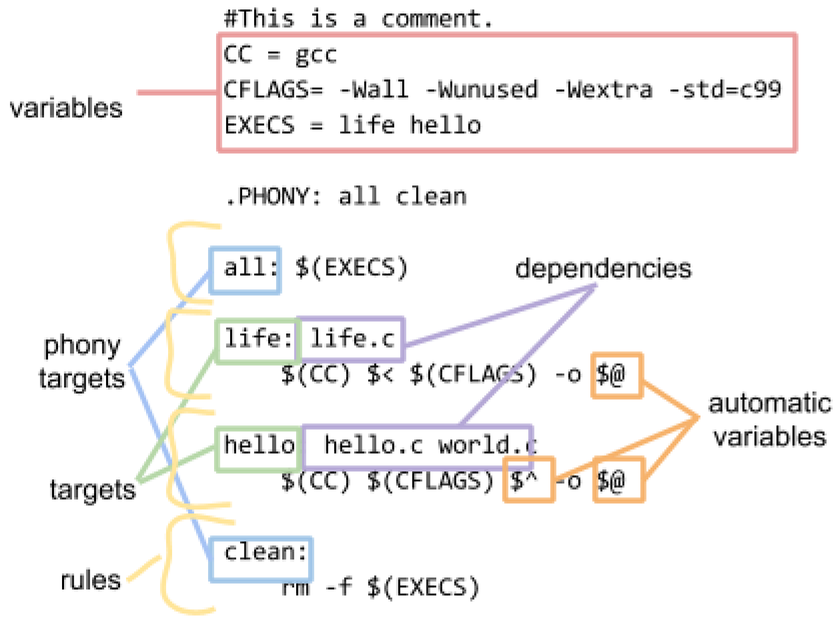
Use the provided **`repltests`** script (usage: **`./repltests quiet_repl loud_repl`**) to verify that **`quiet_repl`** and **`loud_repl`** have the correct functionality. Note that if your **`test`** rule is implemented correctly, you should be able to run **`make test`** to also test functionality.

4 Useful Definitions

- *Target*: a label that denotes commands to run
- *Phony target*: a target that doesn't correspond to the name of any build output.
- *Rule*: set of targets and commands

- *Dependency*: either file name(s) or the name of another target upon which this target depends
- *Automatic variable*: variables represent a target's rule, using a fun symbolic syntax (see [list of automatic variables](#))

Example:



```
#This is a comment.
CC = gcc
CFLAGS= -Wall -Wunused -Wextra -std=c99
EXECS = life hello

.PHONY: all clean

all: $(EXECS)

life: life.c
	$(CC) $< $(CFLAGS) -o $@

hello: hello.c world.c
	$(CC) $(CFLAGS) $^ -o $@

clean:
	rm -f $(EXECS)
```

5 Getting Checked Off

Once you've completed the lab, go to lab hours to get checked off. For this lab, we will have a separate SignMeUp queue for lab checkoffs. To keep the lab checkoff line moving as fast as possible, you are not eligible for immediate help if a TA discovers a problem with your Makefile during checkoff; you will have to sign up for the normal line. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.