CpSc 1111 Lab 10
Makefiles

Overview

By the end of the lab, you will be able to:

- understand the relationship between the `make` command and makefiles
- create a simple makefile typical for a CpSc 1110 project
- understand how makefiles are used in real-world projects

Background Information

If you were not sick of typing and re-typing the command to compile your programming assignment, you probably will soon tire of typing longer lines than that with each successive programming assignment. Maybe you have discovered how the up arrow key ▲ on the keyboard will scroll through your command history, but even that can become cumbersome. In lab today, you will learn how to use a very important tool called **`make`**. It helps automate tasks that would otherwise be repetitive.

- -

Makefiles

Your "Programming in C" book explains makefiles on pages 341-342. A Makefile is a file, called either `Makefile` or `makefile` (can start with an upper case 'M' or lower case 'm') that contains a list of files and their dependencies along with commands that you would otherwise type at the command prompt. When you type `make` at the command prompt, that file called `Makefile` (or `makefile`) is sought in the current directory, and if found, the command for the first target in that file will be run. Your makefile can have multiple targets, as you will see later with this lab. In order to execute the other targets, you would type: `make <target_name>`

Real world programs usually consist of many files that are all compiled and linked together to form one executable. If a programmer makes a change or fixes a bug in just one of those files, by typing `make` (or `make compile` if the target is named `compile`) at the command prompt, only that file that was modified will be recompiled rather than recompiling every single file. The `make` utility can tell which source files need to be compiled based on the modification times of the files. If `make` finds that your source (.c) file is newer than the corresponding executable file (e.g., a.out) or object (.o) file, it automatically recompiles the source file to create a new executable or object file.

Makefiles can be much more complicated than what you will be using with this lab, as the compile commands become more complicated with bigger programs consisting of multiple files. But, even for what you will be doing in this course, you will find makefiles to be very useful and should practice using them throughout the rest of the semester.

```
lab6: lab6.c
      gcc -Wall -o lab6 lab6.c
```

The example above shows a possible, very simple, makefile that you could have used with one of the labs. If you had put those two lines into a file called `Makefile`, then whenever you needed to compile your program, you would simply just have typed `make` (or `make lab6`) at the command prompt. Let's take a closer look at those two lines:

Target name
How will you refer to this target?

Dependencies
What files or targets does the target name depend on?

```
lab6: lab6.c
      gcc -Wall -o lab6 lab6.c
```

Tab Character - NOT spaces!

Commands
What command(s) should be run when this target is called?

Lab Assignment

1. While logged in to one of the School of Computing servers (adas, joeys, or koalas e.g. koala1.cs.clemson.edu) copy two files to your working directory by either using the `cp` command or the `wget` command:

   ```
   cp /group/course/cpsc111/public_html/F18Labs/lab10/* .
   ```

   Don't forget the . (dot) at the end. That copy command above says the following: copy all the files (the wildcard `*` means all) at that location `/group/course/cpsc111/public_html/F18Labs/lab10/`) to your current location (indicated by the . (dot)).

   Or, you can do the following:

   ```
   wget https://www.cs.clemson.edu/course/cpsc111/F18Labs/lab10/reverse_echo.c
   wget https://www.cs.clemson.edu/course/cpsc111/F18Labs/lab10/README
   ```

2. Take a look at the README file for a description of what the reverse_echo.c program is supposed to do. There is some code that you will need to complete for that program to work.

3. Take a look at the reverse_echo.c file. You will notice pretty quickly that the program uses command line arguments. A program that uses command line arguments allows you to pass arguments from the command line; which means – when you run the program, you don't just only type the executable name, such as: .

   ```
   ./reverse_echo
   ```

   but the executable name along with some arguments, such as:

   ```
   ./reverse_echo testing 1 2 3
   ```

   In order to do this, the `main()` function signature changes from what you are used to:

   ```
   int main(void)
   ```

   to this:

   ```
   int main(int argc, char *argv[])
   ```

   The first argument `argc` holds the number of items typed at the command prompt, including the executable name. So for the example above, `./reverse_echo testing 1 2 3` argc would have the value 5 because the executable name + 4 arguments = 5.

   The second argument `char *argv[]` is an array – for now, you can think of it as an array that holds each of those items that were entered at the command prompt (even though that is not exactly correct, it's actually an array of character pointers, but it's a simplified explanation for now). So, you can picture that for that same example above, in memory, the array would look like the following (using our simplified model):

argv[0]	./reverse_echo
argv[1]	Testing
argv[2]	1
argv[3]	2
argv[4]	3

4. Add code to the for loop in the `reverse_echo.c` file so that the program behaves according to the description in the README file. Don't forget to add the appropriate header information; reminder box below:

Reminder About Formatting and Comments

- The top of your file should have a header comment, which should contain:
  - Your name
  - Course and semester
  - Lab number
  - Brief description about what the program does
  - Any other helpful information that you think would be good to have.

5. Compile your modified program by typing the following:
   `gcc -Wall -o reverse_echo reverse_echo.c`

6. Probably after typing the above compile command even once, you will realize that typing just simply `make` would be a whole lot easier. So now, even if you haven't gotten your program to work yet, (** especially if you haven't gotten your program to work yet **) create a makefile. Using your editor of choice, create a file called `Makefile` and in it, put the following:
   [NOTE: Don't forget that in front of each command, (the lines colored red below), you MUST hit the tab key, not the spacebar. If you try to copy and paste this code from below, that may not work either – some editors will add spaces in place of the tabs.]

```
# my first makefile

reverse_echo:
   gcc -Wall -o reverse_echo reverse_echo.c

run: reverse_echo
   ./reverse_echo

clean:
   rm reverse_echo
   rm -f output.txt

test: reverse_echo
    echo Testing a few examples:
   @./reverse_echo testing 1 2 3
   ./reverse_echo testing again
   ./reverse_echo hello world
   ./reverse_echo "single string"
```

7. Once you are done with your makefile, save and close your text editor. To compile your program, you can just simply type `make` because the "reverse_echo" target is listed first. If it wasn't listed first, then you would type:
   `make reverse_echo`

   So, what just happened? We defined a ***target*** called `reverse_echo` (could be any label, but you get some added benefits by using the same name for the label and the executable). That target has one ***dependency***: `reverse_echo.c` (don't compile if that file doesn't exist or if it hasn't changed since the last time you compiled). The second line has the command associated with `reverse_echo`. Then, when you ran `make`, the command automatically looked for a file named `Makefile` in the current directory (which it found) and ran the command for the first target (the `reverse_echo` target). It then printed out the command that it was about to execute, then ran the command.

8. Before testing out the rest of the targets, complete your reverse_echo.c program if it isn't done yet. Use the makefile by typing `make` each time you compile your program. Once your program works, continue with the following steps.

9. Test out the second target in your makefile – the `run` target. Type `make run` at the command prompt. Notice that it depends on the results of the compile; you wouldn't be able to run the program until it compiles. Also notice that when it is run, it just prints out the value of `argc` -- why is that all it does? Now type the following: `make run > output.txt` What happened? Type `ls` to make sure that output.txt file is there; view the contents – what are at least 2 different ways that you can view the contents?

10. The next target is `clean`. This target doesn't depend on anything; you can run this at any point. The purpose of this target is to remove whatever files you specify with the `rm` commands – in our case, it will remove the executable and the output.txt files, if they exist. Type `make clean` and verify that your executable and your output.txt files are both gone.

11. The last target to try out is `test`, which includes some test cases. This directive depends on the results of the compile (you cannot run the program if it did not compile), so if you removed the executable in the above step, re-compile the program (using the makefile). Just as you saw with the `clean` target, this one has multiple commands associated with it. Notice the second line is preceded with an @ symbol; this suppresses the printing of the command before it runs. Try it out and you'll see that command is not printed out when it runs.

Turn In Work

1. Before turning in your assignment, make sure you have followed all of the instructions stated in this assignment and any additional instructions given by your lab instructor(s). Always test, test, and retest that your program compiles and runs successfully on our Unix machines before submitting it.
2. Show your TA that you completed the assignment. Then submit your `reverse_echo.c` program AND your `Makefile` to the handin page: http://handin.cs.clemson.edu. ***Don't forget to always check on the handin page that your submission worked. You can go to your bucket to see what is there.***

Grading Rubric

For this lab, points will be based on the following:

Functionality	75 (compiles without warnings and test cases work with your makefile)
Formatting	10
Inclusion of Makefile	15

NOTE: there could be other possible point deductions for things not listed, such as (where applicable) global variables, use of break not in switch statements, naming the file incorrectly, etc.