CMPE-380: Applied Programming

Laboratory Exercise 05

Makefiles

Table of Contents

Makefiles

Pre-Lab – 10 pts

In this pre-lab session, we will review Makefiles by going through a couple of straightforward examples. Please review the class presentation "make" slides and create a directory named "lab05" in your home directory and change to that directory.

What is Makefile

Makefiles help to automate certain tasks in software projects, like compiling proper branches of the code repository, organizing output files, cleaning specific directories and running third party tools to ensure that the project meets certain quality standards. Make is a rule-based language and used to build, install, and uninstall software. It automatically finds which files need to be built and rebuilds a minimum set of files each time. Make is a general-purpose dependency-tracking software built into Unix/Linux systems. Make is not specific to the C language.

Make files are traditionally named "Makefile" (with an upper-case M). The make program defaults to calling the "Makefile" if no other parameters are specified (e.g. make). In addition, the default behavior of a "Makefile" is to run the "all" target, see the class notes for more information. Specific make files can be executed using the **-file** option

e.g.: **make -f myMakefile**.

Why We Need It?

Consider a software repository with hundreds of developers, thousands of files and millions of lines of code. It can literally take days to recompile the entire code base. What is needed is a general-purpose tool to identify and compile the files that have changed while leaving the unchanged files alone. Make "commands" are just the command line instructions you would execute by hand but integrated into a rule base engine.

CMPE380 Makefile Requirements

For all your CMPE380 home works, most Makefiles will need to have a all, clean, mem and help options.

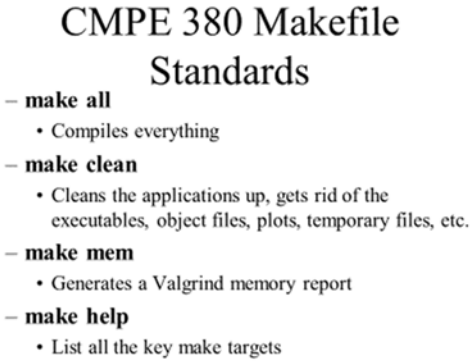

Figure 1 CMPE380 Makefile standards

All cmpe380 make files shall include the following defaults:

1) Compiler: **gcc**
2) Compiler Flags: **-O1 -g -Wall -std=c99 -pedantic**
3) Valgrind command: **VALGRIND = valgrind --tool=memcheck --leak-check=yes --track-origins=yes**
4) Silent and phony options

Create A Simple Make file

Using the class notes as a guide, create a simple make file called "Makefile1", which will compile "fib.c" to an object and link only when needed. Your make file must do the following:

1) Support: all, clean, mem and help entry points
2) Compile the fib.c code to an object only when necessary.
3) Link the fib.o code to the fib binary only when necessary
4) Compile/link the fib.c code only when necessary for the all and mem cases.
5) Print a friendly message when compiling code
6) Use macro variables for all mainline make code (all, mem and clean entry points).
7) Your makefile may use a two step compile and link command:

   e.g. gcc cfile -c and gcc objfile -o binname

To run your make file execute: make -f Makefile1 <entry point>

Note: Many PC editors replace the tab key with spaces. Real Linux editors like **vi** and **vim** use the real 0x07 TAB character required by Make. Notepad++ can be configured to use real tab characters. If you receive an "missing separator" error you are most likely not using real TAB characters.

Your TA will conduct the following tests:

1) Visually inspect your Makefile1 code for proper variables, compile options, silent and phony usage
2) Run: make -f Makefile1 clean (twice) no errors, fib binary is removed
3) Run: make -f Makefile1 mem code should automatically build and run Valgrind
4) Run: make -f Makefile1 mem code should ONLY run Valgrind
5) Run: make -f Makefile1 all code should do nothing
6) "touch" the fib.c file and run: make -f Makefile1 code should automatically build
7) Run: make -f Makefile1 help print friendly help messages

Interactive Exercises – 30

Special Makefile variables

In this exercise will write a make file to illustrate each of the special make file macro variables. A templated make file called "Makefile2" is provided. To run the make file execute:

make -f Makefile2 <cmd> where cmd will be the command to execute.

Redirect the output from the make file and other commands to **exercise.txt** and show the results to your TA when you are finished. You will be documenting your comments in exercise.txt, be sure to clearly indicate what step you are commenting on. You will want to test each stage of your make file before redirecting the output into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

Note: The provided files a.c, b.c and c.c must be compiled/linked to a SINGLE program.

DO NOT USE **.SILENT:** IN THESE Makefiles

1) Add the following to your make file:

```
target1 target2:
        @echo "Target name is '$@'"
```

2) Run: make -f Makefile2 target1 > exercise.txt
3) Run: make -f Makefile2 target2 >> exercise.txt
4) What was printed in both cases? Add your conclusion about the $@ variable to exercise.txt.

5) Add the following to your make file

```
bin1 : a.c b.c c.c
        @echo "Dependency change for bin1 is '$?'"
        gcc a.c b.c c.c -o bin1          (notice 3 c files produce 1 binary)
```

6) Run: touch a.c b.c c.c which will set the file date/time stamps to the current date/time
7) Run: make -f Makefile2 bin1 >> exercise.txt 2>&1
8) Re-run: make -f Makefile2 bin1 >> exercise.txt 2>&1
9) Run: touch b.c c.c
10) Re-run: make -f Makefile2 bin1 >> exercise.txt 2>&1
11) Summarize your conclusion about the $? variable in exercise.txt.

12) Add the following to your make file

```
bin2 : a.o b.o c.o
        @echo "Linking '$^' to '$@'"
        @echo "The first dependency of '$^' is '$<'"
        gcc $^ -o $@
```

13) Run: make -f Makefile2 bin2 >> exercise.txt 2>&1
14) Notice your make file doesn't have a rule to convert .c files to .o files. Are .o files being generated? Comment on the compile lines in exercise.txt
15) Run: touch b.c
16) Re-Run: make -f Makefile2 bin2 >> exercise.txt 2>&1
17) Comment in exercise.txt on what was compiled, is it what you expected?

18) Run: make -f Makefile2 clean
19) Add a ".c.o" rule with @echo "Compiling '$*'" after the .c.o: command. Notice your rule doesn't actually compile any code.
20) Re-Run: make -f Makefile2 bin2 >> exercise.txt 2>&1
21) Comment in exercise.txt on what was compiled, is it what you expected? Why didn't the C file compile?

22) Run: make -f Makefile2 clean

23) Add the following compile line to your ".c.o" rule after the echo command: "gcc $*.c -c"

24) Re-Run: make -f Makefile2 bin2 >> exercise.txt 2>&1

25) Comment in exercise.txt on what was compiled, is it what you expected? Did the C files compile this time?

26) Comment out the .c.o:, echo lines and gcc lines with a "#"

27) Add the following to your make file

```
%.o : %.c
        @echo "Dependency change is '$*'"
        gcc $*.c -c
```

28) Run: make -f Makefile2 clean >> exercise.txt 2>&1

29) Run: make -f Makefile2 bin2 >> exercise.txt 2>&1

30) Run: touch b.c

31) Re-Run: make -f Makefile2 bin2 >> exercise.txt 2>&1

32) Comment in exercise.txt on what was compiled, is it what you expected? Did the echo print as expected? What did $* print in each case? What did $^ and $< print?

33) Comment out the %.o:.%c: echo lines and gcc with a "#"

34) Add the following to your make file

```
a.o b.o c.o : a.c b.c c.c
        @echo "Compiling '$*.c'"
        gcc $*.c -c
```

35) Run: make -f Makefile2 clean >> exercise.txt

36) Run: make -f Makefile2 bin2 >> exercise.txt

37) Comment in exercise.txt on what was compiled, is it what you expected?

38) Run: touch b.c

39) Re-Run: make -f Makefile2 bin2 >> exercise.txt

40) Comment in exercise.txt on what was compiled, is it what you expected?

Using Macro variables

Copy your Makefile2 to Makefile3 and then edit Makefile3

1) Comment out the following in Makefile3:

```
a.o b.o c.o : a.c b.c c.c
        @echo "Compiling '$*.c'"
        gcc $*.c -c
```

2) Re-enable the following in Makefile3

```
%.o : %.c
        @echo "Dependency change is '$*'"
        gcc $*.c -c
```

3) Convert all the c source file lists to a single macro variable
4) Use the macro code reviewed in the class notes to convert the list of .c files to a macro list of .o object files
5) Covert your make file to use the object file macro
6) Convert the bin1 and bin2 binary names to use macro variables.
7) Run: make -f Makefile3 clean >> exercise.txt 2>&1
8) Run: make -f Makefile3 bin1 >> exercise.txt 2>&1
9) Run: make -f Makefile3 bin1 >> exercise.txt 2>&1
10) Comment in exercise.txt on what was compiled, is it what you expected?

11) Run: make -f Makefile3 clean >> exercise.txt 2>&1
12) Run: make -f Makefile3 bin2 >> exercise.txt 2>&1
13) Run: make -f Makefile3 bin2 >> exercise.txt 2>&1
14) Comment in exercise.txt on what was compiled, is it what you expected?

15) Run: touch b.c
16) Run: make -f Makefile3 bin2 >> exercise.txt 2>&1

17) Comment in exercise.txt on what was compiled, is it what you expected?
18) Add support for a mem (Valgrind) option, running "bin2" with no options redirecting standard out and error to mem.txt using macro variables. Your mem option MUST build the binary if necessary.
19) Run: make -f Makefile3 clean >> exercise.txt 2>&1
20) Run: make -f Makefile3 mem >> exercise.txt 2>&1
21) Comment in exercise.txt on what was compiled, is it what you expected?

Show your TA your completed exercise.txt and mem.txt files.

Assignment – 60 pts

Objective

Continue implement a linked list abstract data type as a C module and incorporate a Makefile into the development process..

Please note that this assignment is a continuation of previous assignment!

Background: For a review on linked lists study G. Semeraro "Chapter 4: Data Structure" and also N. Parlante "Linked List Basics" (posted in MyCourses)

Program Specification

1. Upload the file **hw5_files.tar** (available in MyCourses) to your **hw5** working directory. The tarball contains:

   header file **LinkedLists.h & ClassErrors.h**

   Test harness:: **TestSearch.c**

   Some simple test files: **oneTest.txt, twoTest.txt** and **fiveTest.txt.** Use as necessary.

   The full test data: **engWords.txt**

   Expected results: **solution.txt**

2. Copy your dynamic array and linked list source code from your previous labs to your lab05 directory. You will add word search features to your existing dynamic array code and your linked list code as identified in the comments in the original code framework. Use the provided test harness SearchList.c to test your code. The calling syntax is:

   ./TestSearch engWords.txt word where **word** is the search word.

Makefile

You must provide a quality Makefile with the following targets: all, test, mem, help and clean. Quality Makefiles utilize .SILENT, .PHONY, use make variables and only compile and/or link the minimum number of files. A Quality Makefile is robust to usage. E.g. Make clean followed by Make mem will automatically rebuild the binary. Your makefile will need to build your dynamic array code, your linked list code and your search test code.

Make Targets;

"all"	-should make TestSearch
"test"	- should run **TestSearch** twice, once with a make macro variable of **SEARCHWORD**, the word should be "**space**" and then with the invalid value "xyzzy" Output should be redirected to "**out.txt**"
"mem"	**-** should run using the macro **SEARCHWORD** with **valgrind** redirected to **mem.txt.** **Note: this may not take longer than 5 minutes to run**
help, clean	- should do the normal things

Note: Going forward, all assignments will require a quality makefile.

Analysis

Write an **analysis.txt** summarizing your implementation results. Create a tarball **lastName_hw6.tar** (lastName is your last name) with all relevant files and submit it.

Grading Criteria

1. (12 points) Correct implementation of search function, TestSearch
2. (15 points) No memory leaks or access errors
3. (25 points) Correct make file
4. (8 points) Analysis of results concise and clear.

Notes

1. To learn more about doubly linked lists read chapter 4 of G. Semeraro's book (posted onMyCourses). If you use other reference sources list those in your **analysis.txt** file.

Student Name: _____

Laboratory Grading Sheet

Lab05 - Makefiles

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: simple make file	**10**		
Interactive Exercises: Special Variables (inspect code, answers for lines 4/11/21/28/33, etc)	**20**		
Interactive Exercises: Using Variables (inspect code, answers for lines 10/14/17/21, etc)	**10**		
Total	**40**		

You must turn this signed sheet in at the end of lab to receive credit!