MacEwan UNIVERSITY

CMPT 201
Spring 2025

Lab 2 – Introduction to makefiles

GENERAL INFORMATION

- Allocated time: 1 lab
- Due date: May 16 at 11:59 PM on the lab mêskanâs page
- Lab weight: 2% of final grade

OBJECTIVES

- To understand the fundamentals of compilation, execution, and automation using makefiles

NOTES

- Properly acknowledge any help or resource you used (add a comment and/or hyperlink).
- Programs without ID boxes will have marks deducted.

INTRODUCTION

It is not unusual for C programs to grow to hundreds of thousands or millions of lines of source code or LOC, for short. Proper organization and compilation of these files are important. In this lab, we will see how to use a `makefile` to perform separate compilation and to help us automate parts of our testing procedure.

INSTRUCTIONS:

Create a working directory for this lab, named lab2.

In steps 2, 3, 4, and 6, you will create a progression of makefile rules. Each next set of compilation rules should be **added to the existing `makefile`, while previous versions are commented out**. The final `makefile` you submit must run the rules you created in sections 4 and 6. A `makefile` should start with an ID Box similar to source code files.

1. Getting Started

Download the following files from mêskanâs to your local machine: `main.c`, `numMod.c`, `numMod.h`, `numArray.c` and `numArray.h`. Transfer these files to the `lab2` folder on the student server. Navigate to the folder and try compiling this program from the command line. Make sure the current working directory (pwd) is lab2. At the prompt type:

```
gcc -Wall -std=c99 main.c numArray.c numMod.c -o lab2 -lm
```

Run the resulting executable to confirm that it works, before writing your `makefile`.

Remove one of the file names from the compile command, and observe what happens.

2. Simple makefile

Write a `makefile` to compile the `lab2` program, using a single compilation command. For now, your `makefile` will have one rule, but you will modify this file in the next sections. Use the same flags as in section 1. Remove the executable `lab2` that you created in the previous section (**`rm lab2`**), then run **`make`** to test your rule. Comment out this rule (using #) before moving to the next step, but do not delete it; it is part of your submission.

3. makefile with separate compilation

Recall that passing the `-c` parameter on the command line to gcc will not create an executable, but will compile the source file to an object (.o) file. Modify your `makefile` to perform separate compilations by adding targets for your `.o` files. You will use these `.o` files (3 in total) to create the final executable.

Something like the following in your `makefile` should do it (note: you will use the `.c` files from this lab, rather than test.c). Don't forget to include the header files (`.h` extension) of modules used by each `.c` file. Use the compiler flags and linker flags from section 1. Only the compiler rules need compiler flags, and only the linker rules need linker flags.

```
test.o: test.c ...
     gcc ... -c test.c
```

Make sure to update your dependencies in your executable target accordingly. For full marks on this exercise, you should have all your dependencies correctly used. You should have 4 rules in your `makefile` by the time you finish this task. Make sure the program still compiles and produces a working executable before proceeding.

4. makefile with automated testing, creating files testLab2In.txt and correctLab2.txt, main.c modified

In this part you will see how to use `make` to run automated tests.

For this task, first prepare an input file to redirect to your executable, so you do not have to type your test case each time you run it. Make a file with 5 input integers in it, called `testLab2In.txt`. This will be used to generate your 'correct' output file, named `correctLab2.txt` , which is used in your testing rule that you add to your `makefile`.

To make your 'correct' output type:

```
./lab2 < testLab2In.txt > correctLab2.txt
```

Create a rule named **`testing`** in your `makefile`. This rule executes two commands:

- One to produce the testing output from your lab2 executable; call the output file `testLab2Out.txt`. Note: input will be coming from `testLab2In.txt` and output will be redirected to `testLab2Out.txt`
- One where you `diff` the result file `testLab2Out.txt` with the verified correct output file `correctLab2.txt`.

Here is a reminder of how the **`diff`** command is used:

```
diff file1_to_compare.txt file2_to_compare.txt
```

Notice that **`diff`** produces no output if the contents of the compared files are identical.

Change the `main.c` file by changing the first output line to `++Array Manipulation Program++`. Try out your automated test on the modified executable and verify that the test fails. If the diff command is preceded by a dash (**`-diff`**), the make rule is not interrupted when a diff error occurs, and make continues to the next test command, if there is one.

If you run **`make testing`** after making a change to your program and see no output, you know your changes did not affect the 'correct' output. You could add any number of tests to the `testing` rule and running **`make testing`** ensures that nothing that worked previously has been broken by recent changes.

5. README text file

Accidentally breaking previously working code is called a regression and catching regressions early is important. Create a text file named `README.txt` in the `lab2` directory, which contains your name, the lab number and your explanation for why it is important to catch a regression as quickly as possible.

6. Adding macros to makefile

Make supports macros, also called variables. Some common macros have standardized names:

CC is the name of the compiler

CFLAGS contains compiler flags, e.g. `CFLAGS = -Wall -std=c99`

LDLIBS contains libraries to be included when linking, e.g. `LDLIBS = -lm` (links the math library)

Special macros begin with a dollar sign and do not need to be surrounded by parentheses. See http://www.cprogramming.com/tutorial/makefiles_continued.html for more information. They can be used to avoid repeating target names and dependencies:

$@ is the name of the target, often used after the -o flag.

$^ stands for all dependencies, which is useful for linking rules.

$< stands for the first dependency, which is useful after the -c flag in compilation rules.

Define the CC, CFLAGS, and LDLIBS macros in your `makefile` with the appropriate values. Comment out the make rules that were added in section 3, and replace them with new rules that use these

macros and the listed special macros wherever possible. This final version must be the one that runs when **`make lab2`** is called from the command line.

Here is an example of a compilation rule that uses macros:

```
test.o: test.c ...
	$(CC) $(CFLAGS) -c $<
```

ATTACHED FILES

- `main.c`
- `numArray.c`
- `numArray.h`
- `numMod.c`
- `numMod.h`

SUBMISSION

For this lab you must submit the tarball named `lab2.tar.gz` of the lab2 directory, created by running **`make tar`**. Running the following command from the `lab2` directory creates a tarball of the `lab2` directory, which unpacks into a directory named `lab2`. The `../lab2.tar.gz` option tells tar to create a tarball named `lab2.tar.gz` in the parent directory. The `-C ../` option changes the working directory to the parent directory, from where tar compresses the contents of the `lab2` directory. You will find the `lab2.tar.gz` file in the parent directory of `lab2`.

```
tar -czvf ../lab2.tar.gz -C ../ lab2
```

Before running the **`make tar`** command, your `lab2` directory should only contain the following:

1. `makefile` (should contain rules from sections 2, 3, 4, and 6; comment out compilation rules from sections 2 and 3)
2. all `.c` and `.h` files (`main.c` should contain the changes you made)
3. `correctLab2.txt`
4. `testLab2In.txt`
5. `README.txt`

Your `makefile` must always contain:

- a **`clean`** rule to remove all executables, object files, and files created by running the programs.
- a **`tar`** rule to create the tarball, which unpacks into a directory named for the lab.

Your submitted tarball should only contain the files mentioned above. Marks will be deducted if files do not have the specified names and for submitting executables, object files, or other extra files.

All submitted code and additional information must be written by you. Copying any amount of code or text from another student or from online resources and claiming it as your own is a violation of MacEwan's Student Academic Integrity Policy.

MARKING SCHEME

Items	Mark
`makefile` : simple compilation	1
`makefile` : separate compilation	3
`testing` rule and associated files	2
Modified `main.c` file to produce an error with `diff`	0.5
`README.txt` file	0.5
Use of macros/variables	1
Use of special macros	1
`clean` rule	0.5
`tar` rule	0.5
TOTAL	**10**