Lab 2: Functions and Multi-File Compilation

CPSC 1021

School of COMPUTING Clemson University

Lab Objective

- Use 'make' and 'Makefile' to automate compiling programs
- Practice writing modular programs in C++
- Practice using functions and function overloading in C++

Introduction

In this lab, you will learn how to use the Linux make command to compile your multi-file projects. The first part of this lab is a tutorial that demonstrates the use of 'make' and how to write a 'Makefile'. In the second part, you will write a program that uses functions and function overloading. You will submit the following documents/code files for this assignment: (1) a design document and test plan via the 1021 Canvas page, (2) a C++ program via Gradescope (linked on Canvas), and (3) a Makefile.

Collaboration Policy

Work with your assigned team members from Lab 0 on each problem, but with no more than one or two other students. If you have been assigned to a team of four in Lab 0, split into two pairs; if you have been assigned to a team of three, you can work together as one group.

Please make sure to integrate your team members who – because of COVID requirements – have to participate remotely. You can use the MS Teams video conferencing feature to collaborate with a remote team member. Use the search bar on the top of the MS Teams app to search for your team member with their CU user name, then click the video icon in the top-right corner of the screen to start a video call.

You and your partner(s) can work on all assignments together, but every student has to submit their own files.

Tutorial: Using 'make' to compile the Lab 1 retirement calculator program

Download the source code **Lab1_retirement_calculator.zip** from Canvas. This is the solution to the retirement calculator problem from lab 1. Spend a few minutes with your lab partner to compare the solution code to the code you submitted.

Once your projects get larger and include more source files, it often makes sense to logically separate them into subdirectories to make them easier to manage. Even if you don't split them up, compiling multiple source files into an executable becomes a more involved and error prone process. Especially when multiple files have compiler errors.

Our retirement calculator program is a fairly small program, but it still requires a lot of typing to compile it:

```
g++ -Wall retirementCalc.cpp compoundCalc.cpp -o retirementCalc.out
```

This is where the **make** command is very useful. **make** allows us to define a script called the **Makefile** which contains instructions on compiling our programs. The goal of this assignment is to write a simple Makefile for our retirement calculator project.

Makefile Targets and Dependencies

Before we write our own Makefile, let's go over the basic structure of any Makefile. The following explanation in this sub-section is copied from chapter "*An Introduction to Makefiles*" in "GNU Make: A Program for Directing Recompilation" available here: https://web.mit.edu/gnu/doc/html/make_toc.html, which is a great resource to learn about make.

A simple makefile consists of "rules" with the following shape:

```
target ... : dependencies ...
        command
...
```

A ***target*** is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as `clean' [...].

A ***dependency*** is a file that is used as input to create the target. A target often depends on several files.

A ***command*** is an action that make carries out. A rule may have more than one command, each on its own line. Please note: you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with dependencies and serves to create a target file if any of the dependencies change. However, the rule that specifies commands for the target need not have dependencies. For example, the rule containing the delete command associated with the target 'clean' does not have dependencies.

A ***rule***, then, explains how and when to remake certain files which are the targets of the particular rule. make carries out the commands on the dependencies to create or update the target. A rule can also explain how and when to carry out an action. [...]

A Simple Makefile Example

Now that we've seen the basic structure of a Makefile, let's proceed with a simple example that demonstrates the use of targets, dependencies, and commands.

In the root directory of the code you extracted, create a file named **Makefile**.

```
touch Makefile
```

Note that the file MUST be named **Makefile** or **makefile** for the make command to identify it.

Open the file Makefile you created above in an editor and add the following to it:

```
MESSAGE=HELLO WORLD!

all: build
 @echo "All Done"
build:
  @echo "${MESSAGE}"
```

IMPORTANT: ***Any command 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 thing to notice is that we defined a **bash variable** at the top of the script. Bash variables are simply **string substitutions**. Using the **$() or ${}** operator, we can replace the variable with the defined string. Use **${}** with curly braces if you are in between double quotes. This is a pretty straightforward concept, but it's very powerful. In this instance, we use a variable to define a message we print to the screen.

Our first target is **all**. It has one dependency, **build**, which is the second target in our Makefile. When make encounters a target to execute, it will read the dependency list and perform the following actions:

- If the dependency is another target, **attempt to execute that target**
- If the dependency is a file that matches a rule, **execute that rule**

Our second target is build. It has no dependencies, which is why there is nothing listed after it.

Both **all** and **build** have one command, **echo** followed by a string. echo is a command that just prints its arguments to the terminal. Including the @ symbol in front of the command prevents make from printing the command.

Now open a terminal in the directory of your Makfile and type 'make'. You should see the following on your screen:

```
HELLO WORLD!
All Done!
```

Notice that “HELLO WORLD!” is printed first. That’s because the all: target is **dependent** on the build: target.

By default, make will execute the **all:** target 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
```

Now type ‘make build’ into the terminal. You should see the following:

```
HELLO WORLD!
```

This time, only “HELLO WORLD!“ was printed because we told make to only execute build, which doesn’t have any dependencies.

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 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**. For large projects, this can significantly reduce compile time because only those targets whose dependencies have changed need to be re-compiled.

A Makefile for the retirement calculator program

We will next write a simple Makefile for our retirement calculator project. Our project is quite small with only two .cpp files. Hence, at this stage, we will mostly use our Makefile as a shortcut to run g++ on our source code. This Makefile also serves as an introduction to using make as a build tool.

Replace the code in the above Makefile with the following code (you may want to make a copy of the above Makefile first, but you need to put it in a different directory or change its name to something different than ‘Makefile’):

```
all: retirementCalc.out

run: retirementCalc.out
	./retirementCalc.out

retirementCalc.out:  retirementCalc.cpp  compoundCalc.cpp
	g++ -Wall retirementCalc.cpp compoundCalc.cpp -o retirementCalc.out

clean:
	rm -f retirementCalc.out
```

Again, note that alll commands that belong to a target must be indented **at least one tab character**.

Together with your lab partner, run the following commands and discuss what you observe:

- Run 'make' and again 'make'. What does make report when you run it the second time?
- Make a change to any of your .cpp files, e.g., add an extra line at the end. Run 'make' again. What happens?
- Run 'make run'. What happens?
- Run 'make clean'. What happens?

We can improve our Makefile by replacing key commands in our file, such as g++, with variables. Change your Makefile to the following:

```
COMPILER = g++
OPTIONS = -Wall
PROGRAM = retirementCalc.out

all: $(PROGRAM)

run: $(PROGRAM)
    ./$(PROGRAM)

$(PROGRAM): retirementCalc.cpp compoundCalc.cpp
   $(COMPILER) $(OPTIONS) retirementCalc.cpp compoundCalc.cpp -o $(PROGRAM)

clean:
     rm *.out
```

Now all key settings to compile our program – the compiler we are using, the options for our compile process, and the name of the executable – are neatly listed on top of the Makefile.

Note that COMPILER, OPTIONS, and PROGRAM are just variable names that we've selected to increase readability. make has pre-defined variable names (https://www.gnu.org/software/make/manual/make.html#Implicit-Variables) that are typically used. For example, there is a predefined variable for 'g++' named CXX. Therefore, we could replace all instances of $(COMPILER) with $(CXX) without having to define CXX in our program, but this would be less readable (at least at this stage where we are learning about writing Makefiles).

Advanced Makefiles

The Makefile we wrote today is just scratching the surface of what is possible in terms of using make to automate the build process. Linux has many useful functions to help manipulate files in a project, and we can access these tools from our Makefile. For example, in a large project, we could place all source files in a separate folder "src" and let make automatically find these files. As an example, take a look at the Makefile for the Linux kernel, one of the biggest open-source projects in the world, which has close to 2,000 lines of code(!):
https://github.com/torvalds/linux/blob/master/Makefile

Problem 1: Patient Charges

You have been hired by the administration of a hospital system that seeks to improve its handling of patient charges. Your task is to write a program to calculate a patient's total charges after being treated. The program should first ask if the patient was admitted as an *inpatient* or an *outpatient*. Make sure the user is restricted to those two choices; if the user selects an incorrect choice, your program should prompt the user to enter a correct choice (implement this routine inside main()). If the patient was an inpatient, the following data should be entered:

- The number of days spent in the hospital
- The daily room rate
- Charges for hospital services (food, lab tests, etc.)
- Medication charges

If the patient was an outpatient, the following data should be entered:

- Charges for hospital services (food, lab tests, etc.)
- Medication charges

Use a single, separate function `validateUserInput()` to check that none of the data entered by the user is less than zero. If any of the inputs is less than zero, this function return false, otherwise return true.

Once all data has been entered and validated, the program should use two overloaded functions `calcTotalCharges()` to calculate the patient's total charges. One of the functions should calculate the total charges for inpatients, the other function should calculate the total charges for outpatients.

Sample Run

```
This program will calculate a patient's hospital charges.

Enter I for in-patient or O for out-patient: I

Number of days in the hospital: 3

Daily room rate ($): 120.0

Lab fees and other service charges ($): 125

Medication charges ($): 59.99

Your total hospital bills is $544.99
```

Submission Instructions

- Use the assignment uploaded to Canvas to submit a **design and test document**. This document must include a plain language description of your program in pseudocode and a table showing details for at least five test cases. Pseudocode must be detailed enough so that the solution's main functionality (e.g., variables that need to be declared, functions that need to be written) can be inferred from it.

- Submit your complete and correct program to Gradescope. Your submission must include 5 files:
  - hospitalCharges.cpp (this one includes `main()`)
  - calcTotalCharges.cpp
  - calcTotalCharges.h
  - validateUserInput.cpp
  - validateUserInput.h
- Only submit the above 5 files. Do not submit a Makefile; you will submit a Makefile as part of problem 2 below.
- Make sure your program compiles and runs on the SoC Linux machines; the autograder automatically assigns 0 points to programs that do not compile
- Your program will be checked for the following output (this example uses the input values from the sample run):

```
Your total hospital bills is $544.99
```

- Code style (proper indentation, consistency, readability, use of comments, etc.) will be considered when grading your submission

Problem 2: Makefile for Patient Charges Program

Write a Makefile for the above patient charges program.

Submission Instructions

- Your Makefile needs to include three targets: all, run, and clean
- Your Makefile needs to use variables for the compile command, the compile flags, and the name of the executable file.
- Your Makefile need to generate an executable names 'hospitalCharges.out'
- Make sure to name your file 'Makefile' (with a capital 'M')
- ONLY submit the Makefile. Do NOT submit the .cpp and .h files from problem 1. The autograder will check your Makefile for correctness.