Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Lab 8, March 23/24

In this lab, you will first learn more about `git`. After that, you will get some practice on the `make` utility and learn more about makefile. Next you will be asked to modify a program by organizing it in multiple files. Finally, you will learn `alias` and `.bash_profile`, in order to improve productivity.

Be sure to get help from teaching assistants whenever you have any questions.

1. First, perform the following steps to get started:

   (i) Login to server bluenose.cs.dal.ca via SSH from a CS Teaching Lab computer or from your own computer.

   (ii) Change your current working directory to the `csci2132` directory created in Lab 1.

   (iii) Create a subdirectory named `lab8`.

   (iv) Change your current working directory to this new directory.

2. In Lab 7, we learned how to use `git`, which is a distributed version control system. We learned how to make use of `git` to manage our personal projects.

   Software developers also often use `git` for collaboration. Nowadays, many open-source software projects chose to store their source code in a public `git` repository, and anyone could download the source code from the repository using `git`. Now let us learn how to create a local copy of a remote repository by obtaining the source files of the latest development version of `git` via `git` itself.

   To do this, we make use of the `git clone` command. We also pass the URL of the git repository as an argument. The following page of the official website for `git` gives us the URL of the repository: `http://git-scm.com/downloads`

   Thus, to obtain a copy of the `git` repository, run the following command:

   ```
   git clone https://github.com/git/git
   ```

   This may take a minute or two, depending on the network speed. After the command finishes, enter `ls` in your current working directory. You will discover a new subdirectory named `git`.

3. Let us take a look at what we have got.

   First, let's find out the amount of disk space occupied by this directory. For this we use the command `du`, which means disk usage. When we enter `du -k git`, this will tell us the size of each directory contained in the directory `git`, measured in kilobytes (that's what `-k` is for). Try this. The last line of the output is the total size of the `git` directory, from which we can see that we just downloaded more than 100 megabytes of data.

   Next, enter the directory `git` and run the `ls` command. You will see lots of C source files and headers files, as well as a Makefile.

   Finally, enter `git log` and take a look. You can see the record of the commits made by developers of `git`.

4. In courses in which one big term project that asks students to code up a (system) software prototype will benefit students greatly, the term projects are typically group projects as they require a large number of modules. In many universities, operating systems, software engineering and compiler construction are such courses, though how they are taught often depends on the instructors. Some instructors may require students to collaborate using `git` by creating private depositories for students with the help of the administrator of the UNIX server and giving students instructions on how to make use of them. Sometimes a version control system is not required and thus such private depositories are not available. In such cases, you can still make use of `git`. One option is to request a free educational account from `github`, and find more information on how to make use of it. The URL of requesting such an account can be found here:

   `https://github.com/edu`

   You are not require to do so for this course (in this course we learn a large number of topics that software developers are supposed to know and we are not building one particular software system), but knowing this will give you one more (good) option of collaborating on your projects in the future.

5. **Note for students who enrolled in the labs on Thursday and Friday morning: The remaining steps of this lab require the content from this Friday's lecture. Therefore, you can either perform them on Friday, some time next week by yourself, or in the lab next week; next week's lab will have fewer tasks.**

   Now let us get some experience of using `make`. In the course account, I have set up a directory storing the set of files used in class which is an example of organizing a program with multiple files. Review the slides used in class if you have not done so:

   `http://web.cs.dal.ca/~mhe/csci2132/handouts/largeprogram.pdf`

   Now, make sure that your current working directory is the `lab8` directory. Then copy the following (entire) directory over (recall the `-r` option of the `cp` command):

`/users/faculty/prof2132/public/make`

Use the `ls` command to verify that this step has been performed correctly. Change your current working directory to the newly created `make` directory using the `cd` command.

6. Execute the following sequence of commands to familiarize yourself with the `make` utility. These commands are introduced on page 16 of the above slides. After you enter each command, pay attention to the commands that the `make` utility used to create the targets, and run the target executable files created. Note that there may also be error messages when running `make clean`; think about why.

   (a) `make`

   (b) `make clean`

   (c) `make all`

   (d) `make clean`

   (e) `make decimal2binary`

   (f) `make clean`

   (g) `make hello`

   (h) `make clean`

7. In class, we learned that in order to use `gdb` to debug the executable file generated from multiple source files, we have to use `-g` as an option of all the `gcc` commands in the makefile. It is however tedious to edit the makefile to change all the `gcc` commands, especially if your program contains a large number of modules.

   To be more productive, we can define a variable and use it as the option for `gcc` commands in the makefile. More precisely, edit the `makefile` using `emacs` so that its content becomes:

```
FLAGS=-g -std=c99

all: decimal2binary hello

decimal2binary: decimal2binary.o stack.o
        gcc $(FLAGS) -o decimal2binary decimal2binary.o stack.o

decimal2binary.o: decimal2binary.c stack.h bit.h
        gcc $(FLAGS) -c decimal2binary.c

stack.o: stack.c stack.h bit.h
        gcc $(FLAGS) -c stack.c
```

```
hello: hello.c
        gcc $(FLAGS) -o hello hello.c

clean:
        rm decimal2binary decimal2binary.o stack.o hello
```

Then, make sure that there are no object/executable files (if there are, enter `make clean`). Now, enter the command `make` to see the list of commands that `make` invokes to compile your program.

8. Read the last page of the slides to review what you learned in class regarding what `gdb` command to use in order to set a breakpoint at line 12 of the source file `decimal2binary.c`.

   Open the executable file `decimal2binary` using `gdb`, set a breakpoint at this line, and print the value of the variable `decimal` when the program pauses at this breakpoint.

9. We can use other values for the variable `FLAGS` in the `makefile`. Try the following two values: `FLAGS=-std=c99` and `FLAGS=-O3 -std=c99`. Each time after you change the value of `FLAGS`, save and exit `emacs`, run `make clean` and then `make all`. Observe what commands `make` invokes to compile your program.

10. Recall the mergesort program that we saw in class. This question asks you to split it into two modules and write a makefile for it. To do this, you can make use of the source code available at (this requires the `-std=c99` option to compile):

    ```
    /users/faculty/prof2132/public/lab8/mergesort.c
    ```

    Copy this file to your `lab8` directory, compile it as a C99 program and run it. Then, create a subdirectory for your work.

    In your work, use one module for the main program and the other for the mergesort algorithm. This requires a header file to share the prototype of the mergesort function.

    Test your work using the `make` utility.

11. When working on a large program, you might often need to run the same sequence of commands over and over again. To improve your productivity, one approach is to create an alias for your command or sequence of commands. In Bash shell, this can be done using the `alias` command (it is described on page 303 of the textbook).

    Suppose that you often preform the following sequence of operations: clear the screen, compile `mergesort.c`, and run the compiled code, then you could run the following command

    ```
    alias cm="clear; gcc -std=c99 -o mergesort mergesort.c; ./mergesort"
    ```

Then after that, each time you enter `cm`, the above sequence of commands will be executed. Try this.

Following this, you could create more aliases (with names other than `cm`) for other tasks as well.

12. When you logout of the system and login again later, you will no longer be able to use the aliases defined in the previous session. To make the aliases created in one login session available for later use, we can make use of one feature of the Bash shell.

    When we login, if our default login shell is Bash, then it will look for a file named `.bash_profile` in the home directory. If this file exists, then commands in it will be executed. Thus, if we create such a file in our home directory, and add the aliases commands into it, then the aliases will be available after we login.

    Now let's try this. Go to your home directory first. Then, use `emacs` to create a file named `.bash_profile`. In it you can write down one or more commands. For now, let's only add the command

    ```
    alias cm="clear; gcc -std=c99 -o mergesort mergesort.c; ./mergesort"
    ```

    When you finish, logout and login again. Now, enter the `lab8` directory and run command `cm`.