Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Lab 7, March 16/17

In this lab, you will first learn how to use pointers to print memory addresses of variables. After that, you will learn how to use `git`, a distributed version control system, to manage your code. Next you will be asked to write a program, and make use of the tools that you learned today. Finally, you will be asked to implement randomized quicksort, and this is the optional part of this lab.

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 lab7.

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

2. We learned that pointers store memory addresses. In low level programming, we often access memory by address. Here let's use pointers to print the memory addresses of elements in an array.

   To print a value stored in a pointer, we use the `%p` conversion specification. This will print the value of the pointer in hexadecimal.

```
#include <stdio.h>

#define LEN 10

int main(void) {
  int i;
  int array[LEN];
  int *p;

  for (i = 0; i < LEN; i++) {
    array[i] = i;
  }
```

```
  for (p = &array[0]; p < &array[LEN]; p++) {
    printf("Address: %p   ", p);
    printf("Value: %d\n", *p);
  }

  return 0;
}
```

Understand what the above program does, enter it using `emacs`, and save the program in a file named `addresses.c`. Then, compile and run it a few times, and answer the following question:

Can you tell how many bytes are used to store an `int` variable on bluenose? Why?

3. Verify your answer by modifying the program `addresses.c` to make it print the size of `int` on a separate line before generating the rest of the output.

   Hint: think of the `sizeof` operator.

4. Now, make a copy of `addresses.c` and name it `addresses2.c`. Edit this new file to change the array `array` to a character array. Store characters a, b, ..., j in this array. Further modify this program so that it will compile. Then run this program again to determine the size of `char`, and use `sizeof` to verify your conclusion.

5. Next make `array` be an array of `double` values that store 1, 2, ..., 10. Name the new program `addresses3.c` and repeat the above process.

6. Version control systems are software that can be used to manage the source code, documents, large website, etc., as computer files. They are very useful for software development, especially when a team of developers are working on the same project, so that individual developers can work on different sections of the project and apply their changes to the project using simple commands. Changes to the projects are identified by *version numbers*, e.g. version 1, version 2. Even if you are the only author of a program, they are still useful. They put timestamps to major changes you make to your program. If you wish to discard the current version, and use an old version instead (think of a bad design decision you may have made), you can do so with a version control system.

   There are many version control systems, such as `cvs`, `svn` and `git`. Among them, `git` is a distributed version control system which does not require a centralized server. Thus, it is far more advanced than `cvs` and many other tools. Even if you use it for your personal project without sharing it with collaborators, `git` is still more convenient. Thus, in this lab, we will learn some basic commands of `git`.

   When you start a job in industry, you might be asked to use a different version control system. In most cases, this is because the company does not wish to spend the cost to

change the system that currently works. However, even if this is the case, the experience with one version control system will help you learn a different version control system easily.

To get started with `git`, let's do some configuration. When we use `git` to create one version of our program, we normally write down some documentation in the log file of `git` to indicate what changes we have made. This acts as a useful reminder. On bluenose, by default, the editor that `git` calls to edit the log file is `vim`. Since we have been using `emacs`, we want to change this default behavior. To do this, run the following command (if you use `vi` or `vim`, do not run this command):

```
git config --global core.editor emacs
```

You need only run this command once. The change will persist even if you log out.

7. Like most version control systems, `git` stores files' current and historical data in a *repository*. The command `git init` can create an empty repository in the current directory.

   Make sure that your current directory is `lab7` and run this command.

   You can run this command in any directory that you have read/write/access permissions to create an empty repository for the project that you store in the same directory. Thus, you can easily create multiple repositories for multiple projects.

8. Now let's add one file, and in particular, `addresses.c`, to this repository. This has to be done in two steps. First, you enter a command `git add addresses.c`. This prepares `git` for the next update (more formally, commit) by telling `git` to check `addresses.c` next time we commit changes to the repository.

   After you enter the above command, enter `git commit`. When you enter this command, `git` will check the file `addresses.c` to see if changes have been made to this file since the last time you committed it to the repository. Since this file was not in the repository at all, `git` will store its content in the repository, and then open an editor (say, `emacs`) to ask you to enter some documentation for version 1 of your project. Enter "Version 1: Added addresses.c" as your documentation, save and exit `emacs`.

9. To verify that the above task was performed correctly, enter `git log`. This will display the log file for the project in the current directory that is managed by `git`. You will see the documentation that you just added for version 1.

10. Now let's edit `addresses.c` and add some comments. After that, save and exit the editor. Enter the command `git add addresses.c` again to ask `git` to check this file next time you commit. Then, enter the command `git commit` again. Since you have made changes to `addresses.c`, `git` will create version 2 of your project, and open an editor to ask you to enter documentation. Write down "Version 2: documentation added to addresses.c", save and exit.

Enter `git log` to verify your work.

11. Sometimes we may make a bad design decision and would like to discard the current files and work on a previous version. Let's pretend that we made a bad decision by editing the file `addresses.c` to remove an arbitrary line; save and exit the editor.

    Now, enter `git checkout addresses.c`. After that, check the content of `addresses.c`, you will find that the deleted line is back. You succeeded in pulling version 2 of `addresses.c` out of the repository.

    Next, remove this file by entering `rm addresses.c`. Let's say it's an accident. You can run the same checkout command again to get version 2 of this file back.

12. How to get an older version of a file instead of the version that we just committed? This can also be done with the checkout command, though we need an additional parameter. Run `git log` again. For each commit, there is a long identifier of the form "commit 7671eabae8b7...". Say, the identifier of your first commit starts with 7671ea. You can enter `git checkout 7671ea addresses.c` to get your old, uncommented file. Here 7671ea is the first several characters of this identifier, and you just have to enter enough of these characters to uniquely identify this identifier.

13. One project can contain multiple files. Use the commands that we have learned to add `addresses2.c` and `addresses3.c` into the repository. You can enter the `git add` command twice (with these filenames as arguments) before one single `git commit`. Use the git log command to verify your work.

14. Sometimes you might have added code to a few different files in the same project before committing, and you would like to make sure that all changes will be committed. In this case, the `git diff` command is handy. When you enter this command, `git` will compare the files in your current folder with files in the repository, and show you the changes that have not been staged for the next commit.

    To try this command, first edit both `addresses2.c` and `addresses3.c` to add documentation. Then enter `git diff`. The output generated is similar to the output of `diff` command. Read the output to understand what it means. Commit changes using what you have learned before.

15. Work on programming project 3 on page 256 of the textbook to further learn how to use pointers. This example asks you to write a function that reduces a fraction to lowest terms.

    When you work on this, make use of `git` to commit each major step of your implementation.

16. This question is optional, and you are not required to work on it for midterm / final exams. It is however very useful.

The C textbook describes the quicksort algorithm and shows an implementation on pages 205-209. Read it to learn quicksort.

We know that quicksort is very fast on arrays (in class, we saw an implementation of mergesort, and the instructor briefly compared these two algorithms). However, the version of the quicksort implemented in the textbook does not work well for arrays that are already sorted (or nearly sorted). This does happen frequently in practice. Think about why this is bad.

There are different ways of avoiding this. One approach is, when we choose the partition element (or more formally, pivot), instead of always choosing the element pointed to by `low`, we choose a random element between (and including) elements pointed to by `low` and `high`. See page 206 for the definitions of `low` and `high`. This algorithm is called randomized quicksort.

Now, make use of the random number generators that we learned in Lab 6 to implement randomized quicksort. You can modify the code given in the textbook.