Instructor: Ashvin Goel
Course Number: ECE344

Home
Lecture Notes
Lab Assignments
Discussion (piazza)
Grades (UofT portal)

Operating Systems

ECE344, FALL 2016
UNIVERSITY OF TORONTO

LAB 1: REVIEW OF C, DATA STRUCTURES

Due Date: Sep 26, 2016, 5 pm

In this lab, you will review C by writing some very simple C programs. You will also write some simple data structures. These data structures will be useful for your future labs, so make sure that they work correctly. To help you, we are providing a testing framework for your programs.

READING

All labs in the course will be done in C, for two reasons. First, some of the things we want to study (e.g., implementation of threads in Labs 2 and 3) require low-level manipulation of registers, stacks, pointers that would be awkward (at best) in higher-level, safe languages such as Java. Second, C/C++ are widely used languages, and becoming proficient in them will be useful.

If you are unfamiliar with C, please check the C tutorials resources we have provided.

LAB MACHINES

We will be using the UG Linux machines for grading the labs. Although most or all of this lab should "just work" in many other environments (Cygwin, Solaris, etc.), the course staff will not be able to assist in setting up or debugging problems caused by differences in the environment. If you choose to do development in an unsupported environment, it is *your responsibility* to leave adequate time to port your solution to the supported environment, test it there, and fix any problems that manifest.

ABOUT GIT

Source code management (or revision control) is used extensively in the commercial and open-source software world today for tracking and merging source code changes. To provide you with some experience developing real-world software, we will be using the Git source code management system for the ECE344 labs.

We will be providing detailed Git instructions in these lab handouts. However, if you need more information, many excellent tutorials and online documents about Git are available. Here is a list of Git resources.

You will also be using Git for uploading your code for our automated marking system. So make sure to follow the Git instructions carefully.

SETUP

You will be working individually on each of the ECE344 labs. Login to any of the UG Linux machines (`ug51-ug100`, `ug132-ug180` and `ug201-250`) on the `eecg.utoronto.ca` network, and then follow the instructions below.

1. Start by setting up your personal information for Git, if you have not done it previously. The "--global" option will update the `.gitconfig` file in your home directory.

   ```
   cd
   git config --global user.name "Your Name"
   git config --global user.email you@example.com
   ```

2. Next make sure that your home directory is not accessible to others. This way you can ensure that you code will not be available to others accidentally.

   ```
   cd
   chmod 700 .
   ```

3. Now create a new `ece344` directory and initialize your Git repository in that directory.

   ```
   cd
   mkdir ece344
   cd ece344
   git init
   ```

The last command should show the following:

```
        Initialized empty Git repository in ../ece344/.git/
```

Your Git repository in `.git` will contain all the versions of code that you have committed to the repository.

4. See the status of your repository.

   ```
   git status
   ```

   This command shows the following:

   ```
           On branch master

           Initial commit

           nothing to commit (create/copy files and use "git add" to track)
   ```

   This command shows that you are currently on a branch of the repository called "master". We will generally not be working with branches in these labs, so you don't need to care about branches too much. You have nothing to commit because we have not added any files to the repository yet.

5. Let's add source files to the repository. We will be providing the sources for all labs in the `/cad2/ece344f/src` directory. The source files for this lab are in `warmup.tar`.

   ```
   cd ~/ece344             # you should be in this directory
   tar -xf /cad2/ece344f/src/warmup.tar
   ```

   See the status of the repository again.

   ```
   git status
   ```

   You should see the following:

   ```
           Untracked files:
             (use "git add <file>..." to include in what will be committed)

                   warmup/
   ```

   Git shows that the `warmup` directory has been added to the `ece344` directory, but it is not being tracked currently because it has not been added to the repository.

6. Add the `warmup` directory to the repository.

   ```
   cd ~/ece344             # you should be in this directory
   git add warmup
   ```

   See the status of the repository again.

   ```
   git status
   ```

   You should see the following:

   ```
           ...
           Changes to be committed:
             (use "git rm --cached <file>..." to unstage)

                   new file:   warmup/.gitignore
                   new file:   warmup/Makefile
                   new file:   warmup/common.h
                   new file:   warmup/fact.c
                   ...
   ```

   This output shows that these files are ready (staged) to be committed to the repository. You need to commit them now. The "unstage" text in the output tells you how to avoid committing a file. Don't run that command.

7. Commit the new files to the repository.

   ```
   git commit -m "Initial code for Lab 1"
   ```

   On running `git status`, you should now see:

   ```
           On branch master
           nothing to commit, working directory clean
   ```

   You can see a log of all your commits by running `git log`. This command will show you commits in reverse time order (last commit is shown first). Right now, you will see

   ```
           commit ad254ec06094a006060826d1373220e49bbc7c63
           Author: Your Name
   ```

```
            Date:   Sun Sep 14 13:22:27 2016 -0400

                Initial code for Lab 1
```

Git assigns an ID to each commit that allows you to name (refer to) each commit uniquely. It is important to use descriptive commit messages, as shown in the log output above, because the commit ID is not human readable.

8. Next, we will name or tag this commit so that we can refer to this commit by name.

   ```
   git tag Lab1-start
   ```

   Running `git tag` again (without any other arguments) will show that the `Lab1-start` tag has been added. To see which commit a tag is associated with, run the `git log` command.

   ```
   git log Lab1-start
   ```

   This will show a log of **all** commits until the tag was created. The `Lab1-start` tag is associated with the commit id shown in the first line of the log (this is the last commit). Later, we will be using a similar `Lab1-end` tag to mark the code you will submit for this lab.

9. For now, we are done with the Git commands. We have provided more Git instructions below. You will find them useful as you make changes to your code. If you have trouble with any of these commands (e.g., you made mistakes), you can restart the setup from scratch by first removing the `ece344` directory.

10. Now run `make` for a simple program.

    ```
    $ cd ~/ece344/warmup
    $ make hi
    gcc  -g -Wall -Werror  -c -o hi.o hi.c
    gcc  hi.o -lm -o hi
    $ ./hi
    So far so good.
    $
    ```

SOME SIMPLE PROGRAMS

Let us get started by writing some simple C programs.

Hello

Write the program `hello.c` that prints out the string "Hello world\n".

```
$ make hello
...
$ ./hello
Hello world
```

You can test your program by following the testing instructions. You should get 1 mark for this test.

Loops

Write the program `words.c` that prints out the words from the command line on different lines.

```
$ make words
...
$ ./words To be or not to be. That is the question.
To
be
or
not
to
be.
That
is
the
question.
```

You can test your program by running the tester again. You should get some more marks for passing this test.

Procedure Calls

Write the program `fact.c` that uses recursion to calculate and print the factorial of the positive integer value passed in, or prints the line "Huh?" if no argument is passed in or if the first argument passed is not a positive integer. If the value passed in exceeds 12, it prints "Overflow".

```
$ make fact
$ ./fact one
Huh?
$ ./fact 5
120
$ ./fact 5.1
Huh?
```

Headers, Linking, Structs

C code can be across multiple source files. Typically, a header file (e.g., "foo.h") describes the procedures and variables exported by a source file (e.g., "foo.c"). Each `.c` file is typically compiled into an object file (e.g., "foo.o" and "bar.o") and then all object files are linked together into one executable.

We have provided `point.h`, which defines a type and structure for storing a point's position in 2D space, and which defines the interface to a translate function to move the point to a new location, to determine the distance between points, and to compare points. Your job is to implement these functions in `point.c` so that the test program `test_point.c` works. Do not modify the `point.h` header file or the `test_point.c` program file.

Basic Data Structures: Linked List

Change the `sorted_points.c` file so that it maintains a list of points sorted by their distance from the origin (0.0, 0.0) as defined by the interface in the `sorted_points.h` header file. The linked list implementation should be useful in the future labs.

The simple test in `test_sorted_points.c` should now run.

Basic Data Structures: Hash Table

Change the `wc.c` file so that it counts how often words occur in an array. The interface to this program is defined in the `wc.h` header file. You must use a hash table to implement this program because your hash table implementation can then be used in future labs. Note that you can use any hash key function, including an implementation (of just the hash key function) available from elsewhere.

The simple test in `test_wc.c` should now run. To check if your implementation works, run the `run_small_test_wc` script.

Now test the efficiency of your hash table implementation with a large input file by running the `run_big_test_wc` script. This script will work correctly if your program takes less than 30 seconds. Otherwise, you will need to think about ways to improve your hash table performance.

TESTING YOUR CODE

You can test your code by using our auto-tester program at any time by following the testing instructions.

USING GIT

You should only modify the following files in this lab.

```
fact.c
hello.c
point.c
sorted_points.c
wc.c
words.c
```

You can find the files you have modified by running the `git status` command. You should see the following:

```
        Changes not staged for commit:
          (use "git add <file>..." to update what will be committed)
          (use "git checkout -- <file>..." to discard changes in working directory)

                modified:   fact.c
                modified:   hello.c
                modified:   point.c
                modified:   sorted_points.c
                modified:   wc.c
                modified:   words.c

        Untracked files:
          (use "git add <file>..." to include in what will be committed)

                tester.log
                tester.out
                wc-small.out

        no changes added to commit (use "git add" and/or "git commit -a")
```

The command shows the files that you have modified, and some untracked files. The `tester.log` and `tester.out` files are created if you run the tester program. The `wc-small.out` file is created if you run the `run_small_test_wc` program.

We will only commit the files that you have edited by hand (the modified files), and not the generated files, in Git. To commit the modified files, we need to add (stage) them again, before they can be committed. In Git, every time a file is modified, it needs to be explicitly "added" again before it can be committed (see the helpful text about adding files in the output of the status command). The explicit add avoids mistakes where a modified file, that you weren't planning to commit, is committed by accident.

Run `git add` on all the C files and then commit them.

```
git add *.c
git commit -m "Committing changes for Lab 1"
```

Use the `git status` command again to see the status of your repository. Now it should only show the untracked files. You can tell Git to ignore these untracked files by adding the names of these file in the `.gitignore` file in the `warmup` directory.

We suggest committing your changes frequently so that you can go back to see them if needed. For example, say you want to see the changes made in the commit above. Run the `git log` command, and look for the commit id associated with the message "Committing changes for Lab 1" (the id will be above the message). Say this id is "112b406b37932fb7dd6b63a2154042f4c906a640". Then run the `git show` command on a 4-6 digit prefix of the id (or you can use any number of digits that uniquely identify a commit).

```
git show 112b4
```

The command above will show all the changes made in the commit above.

You can modify files as often as you want, and then follow the `add` and `commit` instructions shown above to commit your changes frequently.

Once you have tested your code, **and committed it** (check that by running `git status`), you can tag the assignment as done.

```
git tag Lab1-end
```

This tag names the last commit. You can use `git log Lab1-end` to show you a log of all commits until the tag was created.

If you want to see all the changes you have made in this lab, you can run the following `git diff` command.

```
git diff Lab1-start Lab1-end
```

Now, if you realize that you want to make additional commits to your code, after adding this tag, then you will need to remove this tag, before adding it again to the last commit. You can remove a tag as follows:

```
git tag -d Lab1-end
```

Now you can add the `Lab1-end` tag again by running the `git tag Lab1-end` command shown earlier.

For your convenience, we have provided the manual pages for the common Git commands. You can also see the options to the git commands by running `git [command] --help`, e.g., `git diff --help`.

CODE SUBMISSION

Make sure to add the `Lab1-end` tag to your local repository as described above. Then, please follow the lab submission instructions.

Please also make sure to test whether your submission succeeded by simulating our automated marker.