

Lab 1: Review of C, Data Structures, Using System Calls

Quick Links: [Home](https://q.utoronto.ca/courses/419441/assignments/syllabus) [Lecture material](https://q.utoronto.ca/courses/419441/pages/lecture-material) [Lab assignments](https://q.utoronto.ca/courses/419441/pages/lab-assignments) [Piazza Discussion](https://piazza.com/utoronto.ca/winter2026/ece353/home) 

Due Date: January 23, 11:59pm EST

Overview

In this lab, you will review C by writing some very simple C programs. You will also be writing some simple data structures that will be useful for your future labs, so make sure that they work correctly. To help you, we are providing a [testing framework](https://q.utoronto.ca/courses/419441/pages/testing-your-lab-code) (<https://q.utoronto.ca/courses/419441/pages/testing-your-lab-code>) for your programs.

Next, you will implement a program that uses several file-system related system calls. System calls provide entry points into the operating system, allowing programs to run operating system code. This part of the lab will provide you experience with writing programs that use operating system functionality extensively.

If you haven't done so already, first make sure to go over the [lab information](https://q.utoronto.ca/courses/419441/pages/lab-assignments) (<https://q.utoronto.ca/courses/419441/pages/lab-assignments>) page.

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 to you.

If you are unfamiliar with C, please check the [C tutorials](https://q.utoronto.ca/courses/419441/pages/lab-assignments#c-lang) (<https://q.utoronto.ca/courses/419441/pages/lab-assignments#c-lang>) resources we have provided.

Lab Machines

We will be using [UG Linux machines](http://www.ece.utoronto.ca/lab-support/home-directories/) (<http://www.ece.utoronto.ca/lab-support/home-directories/>) 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](http://en.wikipedia.org/wiki/Revision_control) (http://en.wikipedia.org/wiki/Revision_control) (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](http://en.wikipedia.org/wiki/Git_(software)) ([http://en.wikipedia.org/wiki/Git_\(software\)](http://en.wikipedia.org/wiki/Git_(software))) source code management system for the ECE353 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. Please see below.

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 ECE353 labs. First, follow the instructions for [getting started with the labs](https://q.utoronto.ca/courses/419441/pages/lab-assignments#started) (<https://q.utoronto.ca/courses/419441/pages/lab-assignments#started>), and login to any of the lab machines. For security, this is a good time to change the password on your account. You can use the passwd command to do so. Now 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 your code will not be available to others accidentally.

```
cd  
chmod 700 .
```

3. Now create a new ece353 directory and initialize your Git repository in that directory.

```
cd  
mkdir ece353  
cd ece353  
git init
```

The last command should show the following:

```
Initialized empty Git repository in ../ece353/.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/ece353s/src` directory on the ECE lab machines. The source files for this lab are in the file `warmup.tar`.

```
cd ~/ece353          # you should be in this directory  
tar -xf /cad2/ece353s/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 `ece353` 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 ~/ece353          # 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 2019 -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 ↗\(http://git-scm.com/book/en/Git-Basics-Tagging\)](http://git-scm.com/book/en/Git-Basics-Tagging) 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 ece353 directory.

10. Now run make for a simple program.

```
$ cd ~/ece353/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](#).

(<https://q.utoronto.ca/courses/419441/pages/testing-your-lab-code>) 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. The words in the command line are available using the argc and argv parameters of main() (see `hi.c`).

```
$ 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 (e.g., 5, 10, 030) passed in as an argument to the program, or prints the line "Huh?" if no argument is passed in, or if the first argument passed is not a positive integer (e.g., 0, negative value, decimal value, or non-integral value). 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: 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.

Furthermore, we will test your code on a large input file, and your program must run in less than 30 seconds. To speed up your hash table implementation, you can use any good hash key function, including an implementation of the hash key function available from elsewhere. However, the rest of the hashing code should be your own implementation. You can choose any hash table size, but a good rule of thumb is to use a size that is roughly twice the total number of elements that are expected to be stored in the hash table.

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.

A Recursive File Copy Program

Now you will be writing a program that will make a copy of a directory and all its files and subdirectories. This program, which we will call `cpr` (copy recursive), will use several file-system related system calls, providing you experience with programmatically using the file system functionality of the operating system. Later in the course, we will study how a file system is designed and implemented, and you will hopefully find this experience beneficial.

To implement the `cpr` program, you will need to use several file and directory related system calls (and some library functions). Your `cpr` program has two main parts, as described below: 1) copy a single file, 2) copy all the files in a directory, and then recursively copy all sub-directories.

Copy a Single File

In Unix, before reading or writing a file, it is necessary to tell the operating system of your intent to do so, a process called *opening* a file. The system then checks whether you have the right to read or write the file, and if so, it returns a non-negative integer called the file descriptor. Then the file is read or written using this descriptor. Finally, when all reads and writes to the file are done, the file descriptor is *closed*.

To see the manual for the `open` system call, type:

```
$ man 2 open
```

Alternatively, you can type "linux man 2 open" in a browser. The "2" stands for a system call. As described in the manual, you will need to include the following files in your source code to use the `open` system call:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Then a file can be opened as shown below. The `pathname` argument is the entire path of the file (e.g., `/a/b/filename`). The `flags` argument specifies whether a file is opened for reading or writing. The `open` system call returns the file descriptor `fd`.

```
int fd, flags;
fd = open(pathname, flags);
```

The descriptor value is negative if there is an error. More generally, in Unix, all system calls return a negative value on error. You should always check the return value of a system call, and if the return value

is negative, perform some type of error handling. For our lab, you should invoke the `syserror()` macro defined in the code that we have provided.

After a file is opened, it is read using the file descriptor `fd` as shown below (note that a read will fail if the `fd` is negative since the open had failed previously). This read system call reads the first 4096 bytes of the file and places the data in the `buf` buffer, and returns the number of bytes that were read. A second call to read will then read the next 4096 bytes of the file. When a read system call has read all the bytes in the file, its return value is 0, indicating that the end of file has been reached.

```
char buf[4096];
int ret;
ret = read(fd, buf, 4096);
```

To copy a file, you will need the following system calls: `open`, `creat`, `read`, `write`, and `close`. You will need to carefully read the manual pages for these system calls to see how they need to be invoked to copy a file.

To check if your file copy function is working correctly, you can use the `diff` program to check whether the source and destination files are the same.

Copy a Directory

Copying a directory involves copying all the files in the directory. To do so, you need to invoke directory-related system calls. In Unix, a directory contains a set of directory entries (also called `dirent` structures). A directory entry refers to a file (or a sub-directory) and contains the name of the file and a number (called `inode` number) that uniquely identifies the file.

Before reading a directory, it needs to be opened using the `opendir` call, similar to opening a file. Then the directory entries can be read one-by-one by using the `readdir` call. Finally, when all directory entries have been read, the directory is closed by using the `closedir` call. These three calls are library routines that are wrappers around lower-level system calls.

To create a directory, you need to use the `mkdir` system call.

Previously, we mentioned that a directory entry structure refers to either a file or a sub-directory. For a file, we need to copy the file, while for a directory, we need to copy it recursively. The directory entry structure does not tell us whether it refers to a file or directory. To find out, you need to use the `stat` system call. This call takes a pathname and returns a `stat` structure that provides various types of information about the file or directory such as its type (file, directory, etc.), permissions, ownership, etc.

The type and the permissions of the file are encoded in the `st_mode` field of the `stat` structure. You will need to use the `S_ISREG` and `S_ISDIR` macros on this field to determine the type of the file/directory (see the man page of `stat`). You can ignore all other types (e.g., symbolic links, etc.). You will also need to extract the permissions of the source file or directory from the `st_mode` field and use the `chmod` system call to set the same permissions on the target file or directory.

To copy a directory, you will need the following library calls: `opendir`, `readdir` and `closedir`, and the following system calls: `mkdir`, `stat` and `chmod`. For the library calls, such as `opendir`, you can read its manual pages with the command `man 3 opendir`.

You need to write your `cpr` program in the file `cpr.c`. To test, whether your program works correctly, run `test_cpr`. It should output "Ok" three times.

Frequently Asked Questions

We have provided answers to various [**frequently asked questions \(FAQ\)**](#) (<https://q.utoronto.ca/courses/419441/pages/lab-assignments#faq>) about the lab. Make sure to go over them. We have provided answers to many questions that students have asked in previous years, so you will save time by going over these answers as you start working on the lab.

Testing Your Code

You can test your code by using our auto-tester program at any time by following the [**testing instructions**](#) (<https://q.utoronto.ca/courses/419441/pages/testing-your-lab-code>).

Using Git

You should only modify the following files in this lab.

```
fact.c  
hello.c  
point.c  
wc.c  
words.c  
cpr.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: wc.c  
modified: words.c  
modified: cpr.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](#)

(<https://q.utoronto.ca/courses/419441/pages/testing-your-lab-code>). 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](#) (<https://q.utoronto.ca/courses/419441/pages/git-resources>). 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](#) (<https://q.utoronto.ca/courses/419441/pages/submitting-lab-code>).

Please also make sure to test whether your submission succeeded by simulating our [automated marker](#) (<https://q.utoronto.ca/courses/419441/pages/testing-your-lab-code#marker>).