EECS 280 Lab 01: Getting Started

Due Sunday, 18 September 2016, 8pm

Before you begin working, your lab instructor will review material from lecture and introduce the standard compilation sequence.

We've also prepared a worksheet which reviews concepts from the first two lectures that you complete as part of this lab.

In this lab, you will connect to CAEN Linux and use basic Linux commands to navigate through files/directories and to write, compile, run, and test a simple program.

Finally, you'll learn to use git, a version control system which can be used to backup your code and keep track of changes you make over time. It's also useful for synchronizing your codebase across multiple different computers.

Requirements

You may work on this lab either individually or in groups of 2-3. Include your name(s) in the comments at the top of the file. Submit the files below on CTools. You do not need to turn in any other files.
If you work in a group, each person must submit a copy in order to receive credit.

Files to submit:

- `main.cpp`
- `eecs280math_tests.cpp`
- `Lab_worksheet01.pdf`

Completion Criteria/Checklist:

To pass this lab, you must finish tasks 1-7.

This checklist will give you an idea of what we look for when grading for completion:

In the code:

- ✓ (Task 3) Change the name in the code.
- ✓ (Task 5) Change the precision for printing floating point values in the unit test.
- ✓ (Task 6) Fix the bug in the main program.

In the worksheet:

- ✓ (Task 7) Answer all worksheet questions.

Here's a brief summary of the files included in this lab. Files you need to turn in are shown with a red background.

`eecs280math.h`	Contains the declarations for functions in the math library. You do not need to edit this file.
`eecs280math.cpp`	Contains implementations for functions in the math library. You do not need to edit this file.
🟥 `eecs280math_tests.cpp`	🟥 Program designed to test the `eecs280math` library.
`test.in`	Contains the input for the main program. You do not need to edit this file.
`test.out.correct`	Contains the correct output for the main program given `test.in`. You do not need to edit this file.
🟥 `main.cpp`	🟥 The main program. Adds two numbers from the user.
🟥 `Lab_worksheet01.pdf`	🟥 Contains questions accompanying the lab activities.

Task 1 - Preliminaries

You Need a CAEN Account

If you don't already have a CAEN account (you probably do. Even 183 students should have one), you will need to work with someone else who does for this lab. Afterward, make your way to the CAEN Hotline in room 1315 of the Duderstadt Center (in the middle area behind the circulation desk) to set up an account ASAP.

Connect to a CAEN Linux Environment

Refer to the "Connecting to CAEN" quick reference on the Google Drive for instructions on several different ways to connect.

Open a Terminal and Navigate to Your Home Directory

If you're using SSH, you already have access to a terminal and should already be in the right place. If you're using VNC, you will need to launch a terminal from the Applications > System Tools menu or right click on the home directory icon on your desktop and select "Open in Terminal".

To check that you are in your home directory, use the `pwd` ("print working directory") command. It should look like "`/home/<your uniqname>`". **Don't type the $ sign below when you enter commands - it's just there to indicate something that should be typed at the terminal.**

```
$ pwd /home/<your uniqname>
```

If you're not in your home directory, you can get there by typing this:

```
$ cd ~
```

Create an `eecs280` Directory

Complete the following steps.

1. Create a new directory called eecs280 with the command:

   ```
   $ mkdir eecs280
   ```

2. Change the working directory to the one you just created. *Protip:* try using the tab key to auto-complete the name of your directory after you type the first few letters.

   ```
   $ cd eecs280
   ```

3. Now check that you're in the correct place using the `pwd` command.

   ```
   $ pwd
   /home/<your uniqname>/eecs280
   ```

For more details on working in Linux and what certain commands do, see the CAEN Linux Quick Reference.

Create a GitLab Account

GitLab is an online hosting service for software repositories. As part of the University of Michigan, you are eligible to use GitLab for free. Open up your web browser and go to:

gitlab.eecs.umich.edu

Once you're there, use your umich username and password to log in. Then, click the "New Project" button near the top right of the page. Name your project something like "eecs280_lab01" and add a description if you like.

Make sure that the visibility level for your project is **private**. This is essential, because otherwise the rest of the world can find it - that would be an honor code violation!

Once the project has been created, you should see a URL for your project. Copy and paste it into a command like this:

```
$ git clone https://gitlab.eecs.umich.edu/uniqname/eecs280_lab01.git
```

This creates a **local copy** of the project, whereas the **remote copy** lives "in the cloud" on GitLab. The idea is that we'll work with the local copy, and if something bad happens there (e.g. your computer catches fire), you can restore from the remote copy.

You may see a warning about an empty repository. That's fine - we'll add files now.

Task 2 - Get the Files and then `git` the Files

For this lab, you'll work with a program we've already started for you. The program prompts the user to input two numbers, adds those numbers, and prints the result. The program itself is trivial, but the main goal is to make sure you're comfortable editing, compiling, running, and testing C++ files. The structure of the program is also designed to illustrate the way programs can be composed from multiple files and demonstrate how the standard compilation sequence works.

1. Change into the directory `git` just created. It has the same name as your project.

   ```
   $ cd eecs280_lab01
   ```

2. Copy the lab 1 starter files from the EECS 280 Google Drive repository to your current directory using the following command:

   ```
   $ wget goo.gl/VZNHKM -O - | tar xzk
   ```

   This might look a little complicated. While it's not crucial you understand exactly how the command works, the basic idea is that we use the `wget` program to download a compressed archive with all the lab files in it. `goo.gl/VZNHKM` is a shortened version of the url at which that archive is found. `-O -` means we want to send the output from `wget` to the standard output stream (note: it's a capital `O`, not a zero), and then the `| tar` passes it on to the `tar` program. `xzk` are the flags we provide `tar` to specify we want to decompress and extract the files from the archive, but not overwrite any existing files with the same names.

   CAREFUL! The k flag (in xzk) specifies that existing files should not be overwritten. If you didn't have it and accidentally ran the command again after working with the files, you could overwrite your work!

   You'll follow a similar process to get the files for each lab throughout the course. In case you are working locally, the files are also attached to the CTools assignment and available on the course Google Drive Repository (see link in header).

2. Verify that everything worked as expected by listing all the files in your current directory. Use the `ls` command to do this. You should see the following files:

   ```
   $ ls
   eecs280math.cpp  eecs280math_tests.cpp  test.in
   eecs280math.h    main.cpp               test.out.correct
   ```

3. Before doing anything else, we want to mention that at any time you can check on the state of affairs using the `git status` command. **It's not a bad habit to do**

this often. If you use it now, you'll see something like the following:

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       eecs280math.cpp
#       eecs280math.h
#       eecs280math_tests.cpp
#       main.cpp
#       test.in
#       test.out.correct
nothing added to commit but untracked files present (use "git add"
to track)
```

4. Now, we want to synchronize all of these files with the local copy of our project using `git`. There are two steps in this process. The first is to **stage** them to be committed. This is like letting `git` know "which files we want to save". Use this command:

   ```
   $ git add -A
   ```

   The `-A` flag indicates we want to add all the files. If you do `git status` now, you'll see the files have been added.

5. The second step is to actually **commit** the changes we've made (i.e. the new files) to our local copy of the project. This is like telling `git` to "actually save the changes". We use the `-m` flag to provide a short message describing the changes we are committing. Use this command:

   ```
   $ git commit -m "Added starter files."
   ```

6. We're not quite done yet, because we've only saved the changes to our **local copy** of the project. We need to **push** the changes to the **remote copy** on GitLab as well:

   ```
   $ git push -u origin master
   ```

   Now the remote copy of your project is safely stored away on GitLab in case anything goes wrong. If your computer was destroyed in a horrible accident, you could simply clone a new local copy onto a different machine just like we did at the start of this tutorial. On subsequent push commands, you won't need to specify the `-u origin master` part.

Task 3 - Editing Code and Compiling

Now that we're all set up with `git` and GitLab, let's start working with the code.

1. First, take a bit of time to browse through the provided code using your favorite text editor. For more information about text editors, please see the "CAEN Linux" quick reference on the Google Drive. Before moving on, at least briefly glance at each of these files:

   `main.cpp` `eecs280math.h` `eecs280math.cpp`

2. In the `main.cpp` file, you'll see a line at the top of the main function that prints a greeting. Use a text editor to change "`YOUR NAME HERE`" to be "`Rumpelstiltskin`".

3. Usually we would wait until making more significant changes to commit our code once again, but for the sake of demonstration let's do it now. Throughout the following commands, you can use `git status` to check how things are going.

   Step 1 is to **stage** the files we would like to commit. Since we've just updated files that `git` is already tracking (they've been added previously), it's easiest to just use the `-u` flag (u for "update"):

   ```
   $ git add -u
   ```

   Step 2 is to actually **commit** the files, with an appropriate message:

   ```
   $ git commit -m "Added name to greeting in main.cpp."
   ```

   If we want, we can also **push** the changes to the remote copy on GitLab.

   ```
   $ git push
   ```

4. Now, compile the main program using g++. To compile the provided code (along the lines of the standard compilation sequence described by your lab instructor), use this command:

   ```
   $ g++ -Wall -Werror -O1 -pedantic main.cpp eecs280math.cpp -o main.exe
   ```

   In addition to the source files `main.cpp` and `eecs280math.cpp`, we specify the name of the output executable file with `-o main.exe`. We also provide several other flags to the compiler to customize its behavior. For the full details, please see the "Compiling" quick reference on the Google Drive.

BEWARE: If you accidentally type something like `-o main.cpp`, you're telling the compiler to put its output in the `main.cpp` source file. The compiler happily does this, overwriting your file with a compiled binary. You can't get your source code back! It's gone forever.

Actually. Let's try it. Go ahead - trust us...

```
$ g++ -Wall -Werror -O1 -pedantic main.cpp eecs280math.cpp -o main.cpp
```

(This is a bad compile command. Don't try this at home.)

Now, open up `main.cpp` in a text editor and observe the destruction. That used to be our code!

5. Good thing we're using version control. Now we'll use `git` to revert back to the last commit we made (before compiling over our poor `main.cpp`).

   To look at previous commits, use the `git log` command:

```
$ git log
commit 5863a40c128aee657aebb671c07cee239af6bb1b
Author: James Allen Juett <jjuett@caen-vnc17.engin.umich.edu>
Date:   Thu May 5 03:14:15 2016 -0400

    Added name to greeting in main.cpp

commit cc0d1d9d7fea37590fd5444c5d825ee3485b3759
Author: James Allen Juett <jjuett@caen-vnc19.engin.umich.edu>
Date:   Thu May 5 01:23:45 2016 -0400

    Added lab files.
```

   We'd like to replace our current version of main.cpp with the one from the commit right after adding the greeting. To do so, use a checkout command with the ID of the commit and the name of the file you want to restore. Either copy-paste the hash or type the first seven characters. Your ID may be different from the one shown here.

```
$ git checkout 5863a40 main.cpp
```

   When you run `git status`, you'll see that main.cpp is no longer marked as having changes.
   Now, open up `main.cpp` again and breathe a sigh of relief!

Task 4 - Running the Program and System Testing

Alright, let's try running the program.

1. Ensure that you have compiled your code correctly into `main.exe`.

   ```
   $ g++ -Wall -Werror -O1 -pedantic main.cpp eecs280math.cpp -o main.exe
   ```

2. Run your program using the command shown below. (Note the `./` before the name of your executable.) Type `3` and `4` for the inputs. Your output should look like below.

   ```
   $ ./main.exe
   Hello, Rumpelstilskin
   Please enter a number: 3

   Please enter another number: 4

   The sum of the numbers is 4
   ```

3. Of course, we can tell from visual inspection that something is wrong here, but let's also look at a way to automate this testing using input and output files.

   Open up the `test.in` file - it contains the same inputs we just typed. Let's feed this file to our program instead of having to type the numbers manually. Try using the following command, which "redirects" input from the `test.in` file to our program:

   ```
   $ ./main.exe < test.in
   Hello, Rumpelstilskin
   Please enter a number:
   Please enter another number:
   The sum of the numbers is 4
   ```

   Next, let's capture the output of the program into an output file. We also do this with redirection. You'll see no output on the screen, because it goes into the `test.out` file instead!

   ```
   $ ./main.exe < test.in > test.out
   ```

   Open up the `test.out` file in a text editor or use the `cat` command to print it to the terminal.

   ```
   $ cat test.out
   ```

```
Hello, Rumpelstilskin
Please enter a number:
Please enter another number:
The sum of the numbers is 4
```

4. We have also provided you with a `test.out.correct` file, which contains the output that should have been produced (i.e. a sum of 7). You can automatically compare your output to the correct output using the `diff` command:

```
$ diff test.out test.out.correct
1c1
< Hello, Rumpelstilskin
---
> Hello, Rumpelstiltskin
4c4
< The sum of the numbers is 4
---
> The sum of the numbers is 7
```

Any differences between the two files are shown. Here we apparently misspelled Rumpelstiltskin (you may have done it correctly) and we have the wrong number. If the files had been the same, `diff` would not have produced any output.

The `sdiff` command works similarly, but can sometimes be easier to read. If the results looks all messed up, you may need to widen your terminal window before running `sdiff`.

```
$ sdiff test.out test.out.correct
Hello, Rumpelstilskin               | Hello, Rumpelstiltskin
Please enter a number:                Please enter a number:
Please enter another number:          Please enter another number:
The sum of the numbers is 4         | The sum of the numbers is
```

Task 5 - Unit Testing

We know that our program isn't working correctly as a whole, but we're not sure what's wrong (or if you already noticed the bug, pretend not to see it for now!).

A decent starting point is to do some unit testing on the `add` function that's used by the `main` function. Once we know whether there seems to be a problem with `add`, we can better focus our debugging efforts.

1. We've provided a unit test in `eecs280math_tests.cpp`. Take a look through the

code to get a feel for what it's testing.

2. Compile and run the test:

```
$ g++ -Wall -Werror -O1 -pedantic eecs280math_tests.cpp
  eecs280math.cpp -o eecs280math_tests.exe

$ ./eecs280math_tests.exe
expected_sum: 0.3
actual_sum: 0.3
eecs280math_tests.exe: eecs280math_tests.cpp:17: void
add_test_basic(): Assertion `expected_sum == actual_sum' failed.
Aborted
```

3. Wat.

4. It seems we have conflicting results. The assertion failed, meaning `expected_sum` and `actual_sum` are not the same. But when we print them, they're both `0.3` as they should be! What's going on?

   We're being tricked by the default (limited) precision with which floating point values are printed. The sums are not actually `0.3`. To change the precision, add this line of code to `eecs280math_tests.cpp` anywhere before the sums are printed:

```
cout << setprecision(20);
```

   Recompile and run the code again. Ah ha.

```
$ g++ -Wall -Werror -O1 -pedantic eecs280math_tests.cpp
  eecs280math.cpp -o eecs280math_tests.exe

$ ./eecs280math_tests.exe
expected_sum: 0.29999999999999998889
actual_sum: 0.30000000000000004441
eecs280math_tests.exe: eecs280math_tests.cpp:17: void
add_test_basic(): Assertion `expected_sum == actual_sum' failed.
Aborted
```

5. It turns out the problem is with the test itself, and not the `add` function (after all, the implementation of add looks pretty straightforward). When comparing floating point values, like `double`s in C++, you never want to just use the `==` operator.

   Instead, we need to allow a bit of tolerance due to "rounding errors" caused by the limited precision of floating point values. We do this by checking whether the magnitude of the difference between two values is very small.

On the accompanying worksheet you will write a function that checks `doubles` for equality using this method. HINT: You can totally use this in your tests for project 1!

Task 6 - Fix the Bug

1. It looks like the add function is probably correct after all. That means we need to look elsewhere for the bug. Go back to `main.cpp` and find it!

2. Fix the bug, compile the code, and run the system test again.

   If `diff` produces no output, it means the system test passed. Well done!

Task 7 - Worksheet

See the accompanying worksheet.

Submit

Turn in the modified `main.cpp`, `eecs280math_tests.cpp`, and a scan/photo/edited copy of the completed worksheet on Canvas. The worksheet should be submitted in pdf format.