ECE/CS 250 – Prof. Bletsch
Recitation #1 – Unix

Objective: This recitation works in concert with the skills you're gaining in the Unix course from Homework 0. Here, you will learn about different computing environments available to you and practice using them. You'll also do some basic file manipulation and text editing and test out a basic C program. You will need these skills so that you can develop C programs, and they're also useful skills if you plan to have a career in computing.

Complete as much of this as you can during recitation. If you run out of time, please complete the rest at home.

Note: The auto-magic power of Eclipse or IntelliJ will not be here to help you. You need to be able to navigate Unix-style systems using the basics: shell interaction, file upload/download, and a plain text editor. In industry, if you can only code if you have an IDE, your career is going to be painfully limited and simple tasks will seem needlessly complex. **Let the Unix flow through you.**

There's lots of IMPORTANT explanation here, not just for this course, but for computing for the rest of your life. Actual tasks you're asked to do are highlighted cyan for your convenience, but read everything. Thanks!

PART 1

1. Your choice of computing environments

We'll use a variety of software in the course, and you have your choice of three ways to get work done. Each have their tradeoffs, and you'll need to become familiar with *at least two* of them in this recitation. Your choices:

I. **Duke Docker Container** from Container Manager (https://cmgr.oit.duke.edu/)
Instantly conjure up a GUI environment accessible via web browser. The environment lives in a *Docker container*, a lightweight and restricted form of virtualization. No admin access or easy means of file transfer, but simple. Does not work well on low-bandwidth connections.

II. **Local tools** on your own computer
With the proper software and setup, your own Windows, Mac, or Linux machine can do what you need. Takes some setup, but once it's working, gives the smoothest UI experience, since you aren't working over a network. But beware, your environment may differ from ours, so for C assignments you'll still need to test on one of the other two before turning in your assignments. *Note: support for this option is best-effort, as computing system vary, and not every member of the teaching staff is familiar with every system.*

III. **login.oit.duke.edu** via SSH (Duke shared Linux cluster)
Login to a machine in the Linux cluster run by Duke OIT via SSH. Access files via your choice of several protocols and techniques, allowing for either remote terminal editors or local editors to edit your code. Most closely mirrors how this kind of computing is done in industry.

In this recitation, *everyone* will get the Duke Docker Container working, then you have your pick of trying out Local Tools or login.oit.duke.edu in the Alternative computing environments section later.

For all of the above, we will be using `git` source code control to both back up our data, track its changes, and submit it for grading.

2. Introducing the Duke Docker Container and `git`

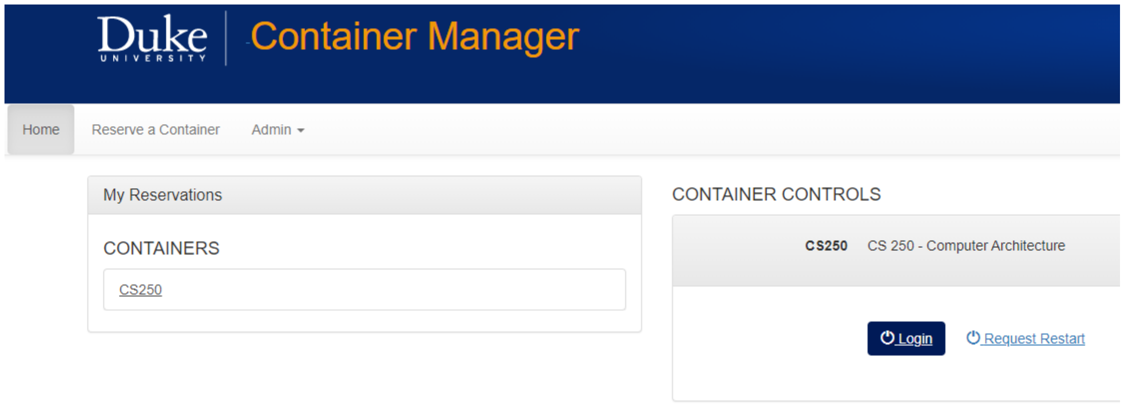
With the help of OIT, ECE/CS 250 provides a container-based online development environment. The containers we provide have all of the programs needed in the class preinstalled.

In addition, we will be using GitLab to distribute test kits for assignments. You can fork an assignment (details later on) to get started. **Make sure that your forked project is private! Not doing so is considered a violation of the Duke Community Standard.** Once you fork an assignment, you can clone the project to your development environment, make changes, commit them and push changes back to GitLab. You are expected to regularly commit changes and push them to GitLab. Because this is a very easy way to keep up-to-date backups, corrupted or lost files will not warrant an extension for homework assignments.

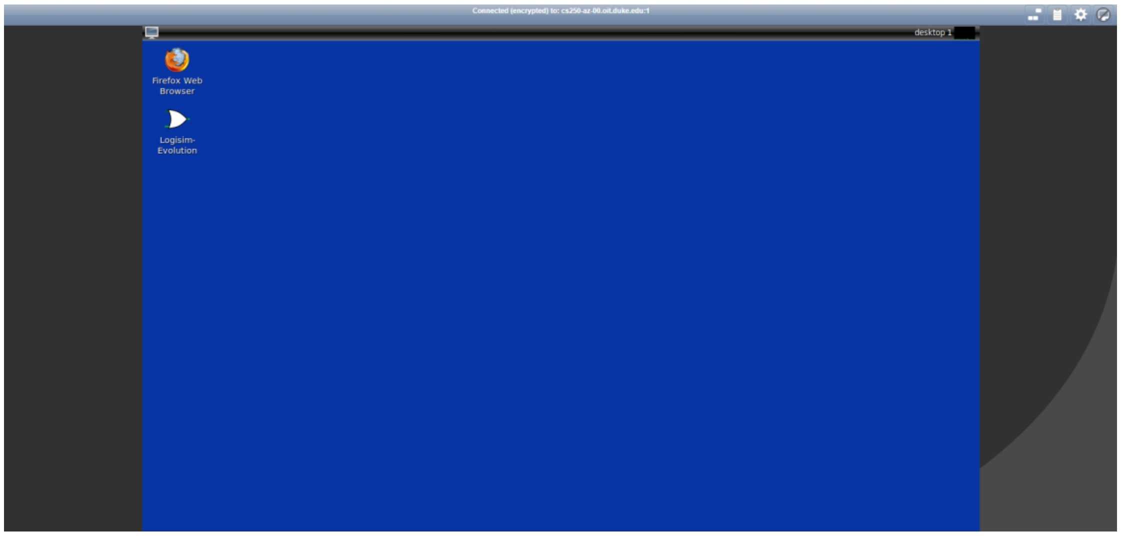
NOTE: File corruption and loss is not a hypothetical scenario, it happens every semester. You should take backing up your work seriously.

Getting an ECE/CS 250 Container Instance

Go to https://cmgr.oit.duke.edu/ and reserve the "CS250 - CS 250 – Computer Architecture" container. Once it's reserved, from now on you should be able to choose "CS250" and hit "Login" to connect to your container. Connect to your container.



If you get a “No session for pid 14” error, it’s okay, just click “OK” to continue. You should see the following screen once you log in:



That’s it! You now have your own ECE/CS 250 container instance for the semester!

Things to Keep in Mind:

- DO NOT bookmark the container, but rather https://cmgr.oit.duke.edu/.
- DO NOT use the ECE/CS 250 containers to run your own programs because extra load will slow down the system for other students. If you need compute for research purposes OIT has other container instances available.
- If your container is hanging use the “Request restart” button on the VM Manage page

Clicking on in the top left of the screen will bring up a menu to select a program to open.

Preinstalled Programs:

- Firefox: for browsing the web.
- Xfce Terminal: preferred terminal to use.
- Visual Studio Code: preferred IDE for writing code.
- QtSpim: program to execute MIPS code for MIPS assignments (HW2)
- Logisim Evolution: a GUI circuit design program for the digital logic and processor assignments (HW3, HW4).

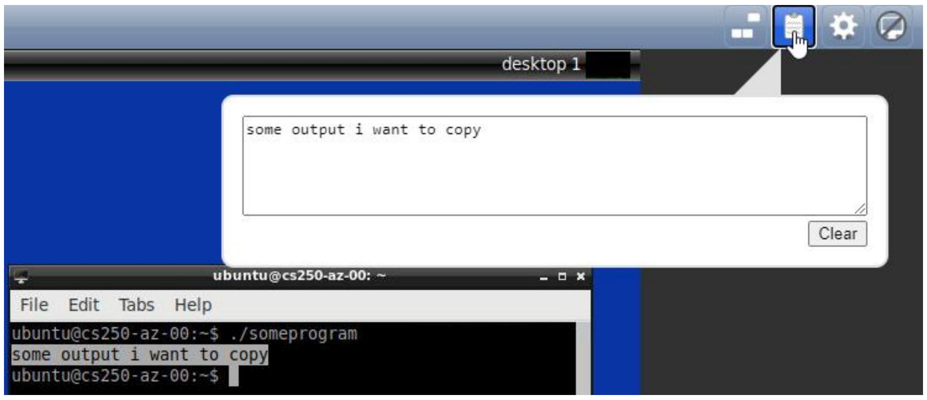
You can drag and drop programs from the start menu onto the task bar at the top to create quick-launch icons.

About the clipboard

Your docker environment constitutes a separate computing environment from your actual computer. As such, they have separate clipboards. This means that copying text on your real computer will not automatically let you paste into your docker environment.

There is a workaround for this: in the upper right is a clipboard button that opens up a textbox. This textbox represents the state of the clipboard of the docker environment. You can copy/paste into the docker's clipboard buffer.

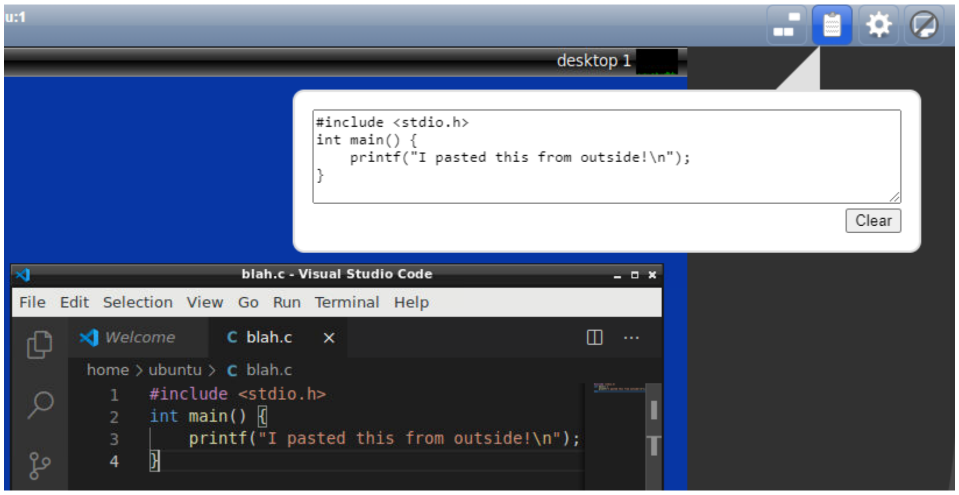
Further, you should note that the terminal in docker (and indeed many terminals) will copy to clipboard automatically upon highlighting – no Ctrl+C or Command+C needed. Below is a screenshot where I've copied some terminal text, then accessed the clipboard buffer to see it on my real computer:



```
some output i want to copy
```

```
ubuntu@cs250-az-00:~$ ./someprogram
some output i want to copy
ubuntu@cs250-az-00:~$
```

Similarly, I can put content into that box from my real computer then use paste within the docker environment. An example of this:


u:1
desktop 1
Clear
blah.c - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Welcome
blah.c
home › ubuntu › blah.c

```
#include <stdio.h>
int main() {
    printf("I pasted this from outside!\n");
}
```

```
#include <stdio.h>
int main() {
    printf("I pasted this from outside!\n");
}
```

Open Visual Studio Code, start a new file, and practice copying text into and out of your container.

3. Terminal warm-up

Poke around your container environment, then open up a terminal (Xfce Terminal). You'll pick up more in Homework 0, but for now, try out these commands.

Useful Commands

- `ls`: list directory contents
- `pwd`: print name of current/working directory
- `cd path/to/directory`: change the working directory
    - Note, `~` is shorthand for the home directory. For example, the Desktop is at path `~/Desktop`.
- `cd ..`: go up one directory level
- `cp` *`src dst`*: copy files and directories from *`src`* to *`dst`* (use `-a` flag for directories)
- `mv` *`src dst`*: move and rename files by moving from *`src`* to *`dst`*
- `mkdir` *`dir`*: make directory *`dir`*
- `rm` *`filename`*: remove file *`filename`*
- `rm -r` *`dir`*: recursively remove directory *`dir`*
- `touch` *`filename`*: create file with name *`filename`*
- `cat` *`filename`*: print contents of file *`filename`* to the console
- `history`: print previous command

Two things you NEED to do on the command line to survive and thrive

1. **Tab completion**: You can use **tab** to complete directory paths and filenames. For example, try typing `cd ~/Desk` and hit tab. This will autocomplete the path as `~/Desktop/`. If the completion is ambiguous (e.g., "pot<TAB>" when there's "potato.jpg" and "potato.txt", it will complete as much as it can, then beep or flash. Hit tab again for a list of the choices, then type a few characters to disambiguate, and hit tab again.

   Only fools type entire filenames by hand: *Always be tabbing!!*
2. **Arrow history**: Use the **up arrow** to access recently used commands (and down arrow go to the other way, too). This can save a lot of time retyping long commands![1]

READ THE ABOVE AGAIN. PRACTICE DOING IT CONSTANTLY.

NEVER NOT TAB COMPLETE. NEVER RETYPE A COMMAND.

I KNOW THE FUTURE. YOU *WANT* TO PRACTICE THIS.

[1] If you want to be really efficient, you can hit Ctrl+R in the shell, then type part of a command you want to go back to. This is a reverse-search of your command history, basically searching the up arrow for a given string.

4. Git and GitLab

Git is a source code control tool that will allow you to track changes over time. GitLab is a central repository for Git projects; there's an instance of GitLab deployed by the Computer Science department for coursework: https://coursework.cs.duke.edu/

Local Git Setup on ECE/CS 250 Container

First, we need to make sure git is setup properly on your ECE/CS 250 container. Open the terminal and enter the following commands, replacing "NetID" and "Your Name" appropriately:

```
git config --global user.email "NetID@duke.edu"
git config --global user.name "Your Name"
```

We now need to set up SSH keys so you can access GitLab (more info here). An SSH key is a cryptographic pair of data files called the "public key" and "private key"; these files are mathematically related. We provide the public key to anyone Gitlab, then we can use our corresponding private key to login to Gitlab in the future. Don't sweat the details – the tools do most of this for you[2]. To make a pair of keys, run the following command in the terminal, replacing "NetID" appropriately:

```
ssh-keygen -t rsa -b 4096 -C NetID@duke.edu
```

Press enter when prompted to enter a path to save the SSH key and also bypass adding a passphrase by pressing enter. Now run:

```
cat ~/.ssh/id_rsa.pub
```

and copy the output. This is your public key. Navigate to https://coursework.cs.duke.edu/ and sign in using the "Duke Shibboleth Login" option. Click on the profile icon in the upper right corner and select "Settings". Now choose SSH Keys on the left sidebar. Paste your SSH public key and give it a descriptive title such as "ECE/CS 250 Container" and click "Add key".

In general, you may find OIT's GitLab basics guide helpful as you become accustomed to GitLab.

You will need to do the above setup on every system you intend to use git on for this class.

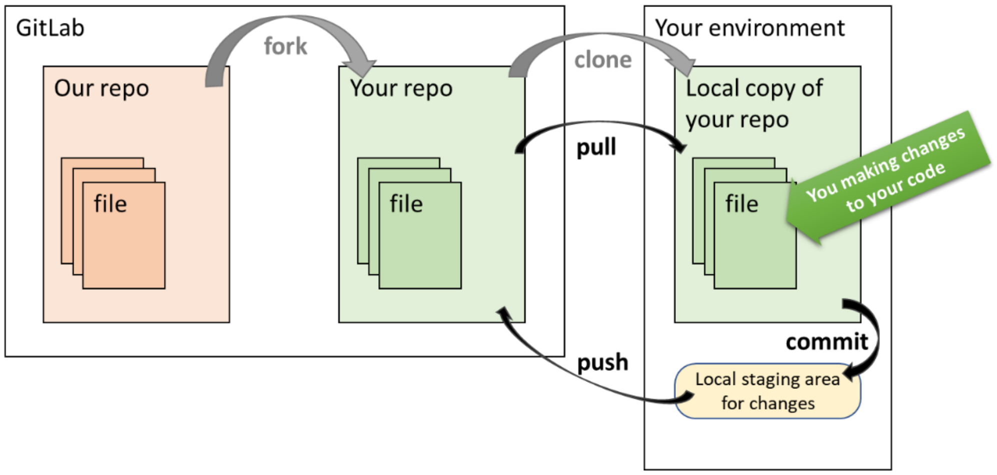
[2] You can learn all about this sort of thing by taking an intro computer security class, such as CS 351 or ECE 560.

Helpful Git Commands

- `git add` *`filename`*: add file *`filename`* to stage for commit
  You'll do this for all your source code files, e.g. `byseven.c` *in Homework 1!*
- `git status`: display the state of the working directory and the staging area
- `git commit -m "`*`MESSAGE`*`"`: commit stages changes with the commit message *`MESSAGE`*
- `git push`: push changes upstream (e.g. to GitLab)
- `git pull`: pull changes from upstream (e.g. from GitLab)

There are many online resources on how to use Git. Git is very powerful and has many cool features. For this class the simple commit and push workflow should be sufficient but if you want to dive deeper into Git try using branches to work on features and merging those features into the master branch! The Git documentation is a good place to start.

The diagram shows the major steps involved in git. One-time steps are shown in grey, whereas steps you do repeatedly during development are in black.



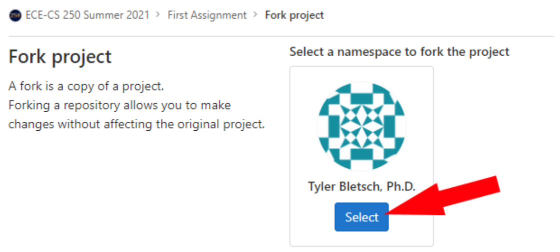
Your First GitLab Assignment

Git step 1: Fork

Now let's fork a project (aka repo) from the ECE-CS 250 Fall 2021 GitLab group, clone the forked project, make changes, commit those changes and push the changes to GitLab. This is the exact same workflow you will use to fork homework assignments, make changes, and backup these changes on GitLab.

Navigate to https://coursework.cs.duke.edu/ and log in if needed. Now navigate to Menu > Groups > Your Groups > ECE-CS 250 Fall 2021. On the sidebar go to Group Overview > Details and select "Subgroups and projects". Here you should click on a project titled "**Getting Started**". NOTE: If you don't see that group listed, then you haven't been invited yet – in that case, use this link to go directly to the repository.

On the upper right click Fork 0 to fork the project and select your own name/NetID when prompted:



ECE-CS 250 Summer 2021 > First Assignment > Fork project

Fork project

A fork is a copy of a project.
Forking a repository allows you to make changes without affecting the original project.

Select a namespace to fork the project

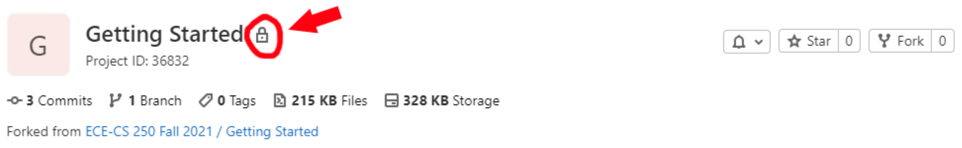
Tyler Bletsch, Ph.D.

Select

Git step 1½: Make your repo private

ALWAYS MAKE SURE PROJECTS ARE PRIVATE!

You should see the 🔒 icon next the project name if it is private:



Getting Started 🔒
Project ID: 36832

3 Commits 1 Branch 0 Tags 215 KB Files 328 KB Storage

Forked from ECE-CS 250 Fall 2021 / Getting Started

If the project is not private, navigate to Settings > General on the left side bar. Then expand the "Visibility, project features, permissions" section and change the "Project visibility" to "Private".

Make sure to fork a project before making changes. DO NOT clone and make changes before forking a project since you will not be able to push changes.

ALWAYS MAKE SURE THE FORKED PROJECT IS PRIVATE! Make sure to make it private if it is not already. Not doing so is considered a violation of the Duke Community Standard.

Git step 2: Clone to your local environment

Now click on Clone in the upper right and copy the link for “Clone with SSH” to clone the project. Open the terminal, navigate to the Desktop (run `cd ~/Desktop`) or where ever you want to store this code, then run:

```
git clone PASTE_LINK_HERE
```

This command will clone the project to your environment. Navigate to your local copy of the project (`cd getting-started`). Use `ls` to see what’s there. What file(s) are present?

Git step 3: Mess with some code!

Let’s compile and run `welcome.c`:

```
g++ -g -o welcome welcome.c
./welcome
```

The first line compiles `welcome.c` into an executable program called `welcome`, and the second line runs the program `welcome`. The “`-o welcome`” part of the first line tells `g++` to create an executable called `welcome`. By default, `gcc` would’ve otherwise created an executable called `a.out`. The “`-g`” tells `g++` to include debug symbols, which will make debugging the program easier (important later on!). In the second line, you may wonder what the deal is with the “`./`”. That tells the terminal to look in the current directory for the file to run, which is necessary for running a program from the current directory[3], but not necessary for reading, moving, renaming, etc. (Your current directory can be referred to with “`.`” and its parent directory can be referred to with “`..`”. So if you type “`cd ..`” that’ll take you to the parent directory.)

Interact with the program and observe what it does.

Open `welcome.c` in Visual Studio Code. You can do this by running Visual Studio Code then opening the program, or by running “`code welcome.c`” on the command line. Change the program to indicate that the so-named person is, in fact, very cool. Compile and run it again to confirm your changes worked.

Git step 4: Commit and push changes

Lastly, and **importantly**, let’s commit this change and push it to GitLab so there’s a backup online. From the `getting-started` directory, run:

```
git add welcome.c                        Adds the changed file to staging
git commit -m "now it’s very cool"       Commits change locally
git push                                 Pushes change to remote gitlab repo
```

[3] The reason for this requirement is security. Imagine a malicious person put a program called “`ls`” in the current directory. When you type `ls`, you might run that program instead of the usual `ls` command. To disambiguate the situation, Linux requires you to be explicit when running a program from the current directory by prefixing it with “`./`”.

Go to the **Getting Started** project on GitLab. You should see the changes reflected there!

Create, compile, and run Hello World

Now let's create a Hello World C program from scratch and execute it.

First open Visual Studio Code and create a file named `hello.c`. Save this file.

Write the following to `hello.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
     printf("Hello World!!\n");
     return EXIT_SUCCESS;
}
```

Compile the program with `gcc` and run it:

```
g++ -o hello hello.c
./hello
```

Add your new `hello.c` program to git, commit it, push it, and confirm it landed in the web interface of GitLab. Don't add the compiled programs `welcome` or `hello` – it's customary for git to hold source code, not compiled programs!

END OF PART 1

If you're in recitation now, please continue and dig into PART 2 until recitation ends.

PART 2

5. Getting acquainted with the tester, hwtest.py

For all the homeworks in this course, you will be provided a student self-test tool, **hwtest.py**.

The tester is not meant to be opaque or mysterious – it's a tool to empower you, because *software testing is the cornerstone of software development*. Let's dig into it!

To get you acquainted with the tool, it's been included in `getting-started`. There is no grade for this, but we'll walk you through using the tool to debug a provided program and submit it to GradeScope for mock grading.

The program we'll be fixing is `square.c`. It takes a single integer as a command line argument and it's supposed to print the square of that argument, but it gets it wrong now. Here's how the tool is *supposed* to work:

```
$ ./square 7
49
```

This is called the *expected* output. However, what happens when you compile and run the program?

What you see is the *actual* output. When expected differs from actual, that means the software isn't meeting the requirements – a bug! This program and its bug are simple, so you could probably just figure it out manually, but let's explore the tester.

The tool runs tests described in the `tests` subdirectory. Key files in there:

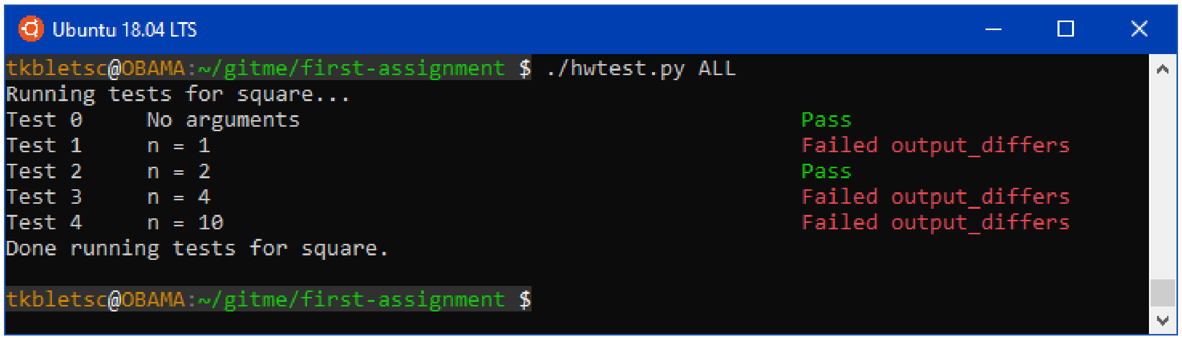
- `tests/settings.json`: The settings for the tester – this describes the tests. We provide it. Tests are divided into test suites, with each suite running against a separate program. In this example, you just have one suite called "`square`" to test the `square` program.
- `tests/<SUITENAME>_expected_<TESTNUM>.txt`: The expected output, provided.
- `tests/<SUITENAME>_actual_<TESTNUM>.txt`: The actual output, generated by hwtest by running *your* program.
- `tests/<SUITENAME>_diff_<TESTNUM>.txt`: The tool will compare the expected and actual outputs. This file is generated to describe how the two differ; it's a `diff`, which is a common UNIX concept.

Let's try out the tester. Run it with "`./hwtest.py`" and it will tell you the arguments:

```
usage: hwtest.py [-h] [-C] [-v] [-t TESTDIR] <SUITE_NAME>

Student auto-tester version 3.0.0.

positional arguments:
```

```
<SUITE_NAME>   A test suite name to run ('square'), or
                'ALL' for all of them.

optional arguments:
-h, --help     show this help message and exit
-C, --clean    Remove generated actual and diff files for chosen suite(s).
-v, --verbose  Verbose mode. Shows the commands executed.
-t TESTDIR     Choose the directory with the tests and test content.
                Default: tests
```

No need to sweat all those options – the most common thing you'll do is just run all the suites, so do that like this:



```
tkbletsc@OBAMA:~/gitme/first-assignment $ ./hwtest.py ALL
Running tests for square...
Test 0      No arguments                               Pass
Test 1      n = 1                                      Failed output_differs
Test 2      n = 2                                      Pass
Test 3      n = 4                                      Failed output_differs
Test 4      n = 10                                     Failed output_differs
Done running tests for square.

tkbletsc@OBAMA:~/gitme/first-assignment $
```

Ah, we're passing two cases and failing three. Let's see what's up with test 1 ("n = 1"). You could use your GUI to navigate there and click files, but you'll become more proficient if you use the command line.

Let's look in tests. Use `cat` to see `square_expected_1.txt` and `square_actual_1.txt` within the `tests` directory *(use tab-completion constantly!)*. Weird bug, huh?

Let's look at the *diff* between these files; this summarizes *how* the files differ. In this program it's obvious, but in later assignments reading a diff will be helpful (otherwise, how can you spot one difference within thousands of lines of output?). Use cat to view `square_diff_1.txt`.

Here, "1c1" which means "on line 1 of the first file, there's a conflict with line 1 of the second file". Then it shows the difference, with "<" prefixed to the first file and ">" prefixed to the second file. The files in question are the expected and actual outputs. You can read more about diff files here.

Look at the other test cases (both passing and failing). Generate a hypothesis about what is happening.

Now open `square.c` in Visual Studio Code and check your hypothesis. Find the bug, fix it, recompile, and confirm that all test cases now pass.

Now submit to GradeScope

The very same tester provided to you is also used to grade your programs automatically. The only difference is that we have more tests than are provided to you, so passing the student tester is not a guarantee of getting a perfect score – you will need to do additional tests on your own. This is true in real life – there's never a level of software testing that's complete or perfect!

The auto-grader runs in the GradeScope environment, which has been configured to hide the results of hidden instructor tests until after the deadline. A GradeScope assignment called **Getting Started** has been created that will examine your modified `square.c`. This is not for a grade – it's just to get you used to the tools. Submit your code to GradeScope (you can either use the GitLab button to directly send your repository or just upload the `square.c` file manually). You should get 10/10 points.

6. Alternative computing environments

The docker container you've been using is nice because it's quick to create and easy to start using, but it's also slow, and a bit limiting in terms of tools available. You're welcome to use this environment, but to give you options, explore either **local tools** or **login.oit.duke.edu**, documented below.

NOTE: No matter which of the methods you use, you should become completely comfortable in your environment and understand every piece of it. If you don't understand something, stop and get help! As you go through the course, you don't want to be doing things "the hard way" without realizing it the entire time.

7. Computing option: Local tools

How you get appropriate local tools depends on your operating system. We've provided basic directions for Windows, Mac, and Linux, but **you will need to do your own research to fully utilize this approach.**

For users of Windows

The computing environment for the course is Linux, specifically Ubuntu Linux. If you're running Windows 10, you can use the Windows Subsystem for Linux (WSL) to create an Ubuntu Linux environment inside of Windows. This isn't a full virtual machine, but is sufficient for our use in most ways.

Note: If you're running another version of Windows or otherwise can't use WSL, you might consider running hypervisor that will allow a full virtual machine, such as Virtual Box. Then you can install Ubuntu 20.04 from scratch (though we can't offer support for this approach).

First, to enable WSL at all, run powershell as administrator (right click powershell in start menu and choose "Run as administrator"). In the powershell, run:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

Then, because this is Windows, you have to restart.

You could install Ubuntu from Windows Store, but instead you can just download this URL: https://aka.ms/wslubuntu2004

Run the downloaded package and it will install an Ubuntu 20.04 environment. Run the Ubuntu bash shell. The following command will install C and related build tools, as well as stuff we use later in the course (Java and spim):

```
sudo apt update
sudo apt install build-essential valgrind make git openjdk-8-jre spim
```

Choose a native Windows text editor to use (Visual Studio Code is a common pick), and install it.

Note: Your virtual Linux home directory is separate from your regular Windows files. To access your Linux files from regular Windows, navigate to `\\WSL$` in Explorer, open dialogs, etc. From there, you can pick Ubuntu, "home", and your Linux account.

If you need to access regular Windows files from Linux, you'll find your C: drive in `/mnt/c` and your Windows user directory in `/mnt/c/Users/<username>`. It's recommended to put your code in your Linux area and access via `\\WSL$` rather than put them in your Windows area and access via `/mnt/c`.

Proceed to the "Put that local computer to use!" section.

For users of Mac OSX

Mac OSX comes with the C compiler and related tools (and if it doesn't, prompts on the command line should walk you through getting it). Note: when compiling with `g++` on Mac, you may get a warning about C++ being treated as C – that's fine and safe to ignore.

NOTE: These directions currently are busted – skip for now. We do need to add a tool called `valgrind` to help debug things later on. To get it, first set up homebrew, a system to allow easy installation of various software. To do so, run the following in a terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Once it's set up, run:

```
brew install --HEAD https://raw.githubusercontent.com/sowson/valgrind/master/valgrind.rb
```

Test `valgrind` on your hello world program by running "`valgrind ./hello`".

If your `valgrind` install doesn't work, just proceed anyway and post on Ed with the issue. We've had issues with `valgrind` on Mac before – worse case, you can use the Docker container or `login.oit.duke.edu`. You'll only need `valgrind` on Homework 1 anyway.

For users of Linux

If you're running Ubuntu Linux 20.04, you're set. Just install some stuff:

```
sudo apt install build-essential valgrind make git openjdk-8-jre spim
```

If you're running a different Linux, you're still probably fine, just install the things above via your native package manager.

Put that local computer to use!

Using your newly configured local machine, do the following:

1. Prepare git as described earlier
2. Check out the `getting-started` repository
3. Modify, compile, and run the `welcome.c` program
4. Run `hwtest.py` and confirm your `square.c` is still passing.

8. Computing option: login.oit.duke.edu (Duke shared Linux cluster)

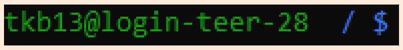
Duke maintains a cluster of x86/Linux machines for your use that you can access through the magic of networking. To connect, we need to use a **secure shell (SSH) client.**

NOTE: If you are not on campus, you will need to connect to campus via VPN. See this page for details and connect to the campus network via VPN before proceeding.

Mac/Linux	Windows
Open the Terminal App. You can find it in the Applications/Utilities folder or by searching in Spotlight for Terminal At the command prompt, type `ssh netID@login.oit.duke.edu` where netID is your Duke NetID. This command initiates a secure shell connection to a Linux machine in the cluster. Enter your password and MFA.	Download and install PuTTY from here. Open a PuTTY terminal window. In the connection screen, for Host Name put `login.oit.duke.edu`. Ensure connection type is SSH and port is `22`. You can save a session for subsequent use by giving it a name and saving the session. Then you can later reload the session by selecting it and clicking "load". Click Open to start the PuTTY session. This will open a Terminal Window and prompt you for your NetID (i.e., login as:), password, MFA.
Congratulations – you are now successfully connected to a remote Linux machine!	

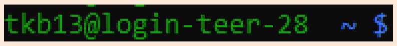
Sidebar if you login via passwordless SSH key

If you've configured SSH keys for auto-login, you'll run into a quirk here based on how Duke runs things – your home directory won't be available by default! You can tell this because you'll appear in the root directory of the system instead of your home directory:


```
tkb13@login-teer-28  / $
```

Note the slash at the end – this means "root". Your files aren't there :-(

To fix this, run "`kinit`" and input your password. Then you can type "`cd`" to go to your home directory. Linux abbreviates your home directory to a tilde (~), so when successful, this will be your prompt:


```
tkb13@login-teer-28  ~ $
```

Note the tilde…you're home! :-)

Via SSH, clone your git repository so we can do work from here.

How to edit

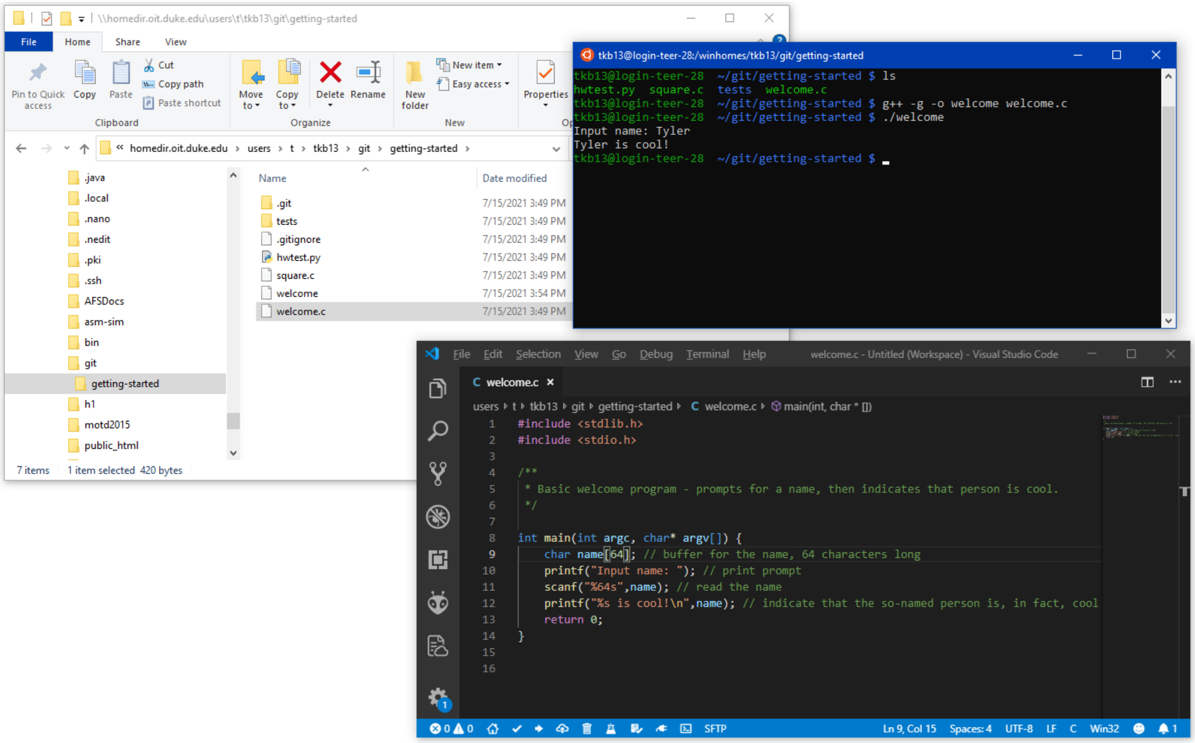
It is certainly possible to use a text editor remotely – there are editors that work entirely from within the terminal (`nano`, `vim`, `emacs` – see appendix if interested) as well as means to see remotely-running editors on your local machine (not covered here).

However, it's easier to use *local* tools to edit *remote* files. We can do that by **mounting** (attaching) our remote home directory to your local computer so that any program can read/write remote files. In this case, every time you hit "save", the file will be sent over the network and saved on the file server, visible immediately to the `login.oit.duke.edu` environment.

Duke is set up to use the "CIFS" protocol (also known as "Windows sharing") to do this. This technique requires you to either be on campus or to use Duke VPN (which makes it like you're on campus). Duke OIT provides documentation on this here:

- General OIT guidance.
- Tutorial for Windows.
- Tutorial for Mac.
- If you run Linux personally or want to mount from your VCM machine, see appendix.

Below is a screenshot on my local Windows machine showing my `getting-started` assignment in my *local* file explorer, my *local* Visual Studio Code, and the *remote* login.oit.duke.edu terminal. Using the procedure appropriate for your OS, get this setup going on your own machine.



```
.git          7/15/2021 3:49 PM
tests         7/15/2021 3:49 PM
.gitignore    7/15/2021 3:49 PM
hwtest.py     7/15/2021 3:49 PM
square.c      7/15/2021 3:49 PM
welcome       7/15/2021 3:54 PM
welcome.c     7/15/2021 3:49 PM
```

tkb13@login-teer-28:/winhomes/tkb13/git/getting-started

```
tkb13@login-teer-28  ~/git/getting-started $ ls
hwtest.py  square.c  tests  welcome.c
tkb13@login-teer-28  ~/git/getting-started $ g++ -g -o welcome welcome.c
tkb13@login-teer-28  ~/git/getting-started $ ./welcome
Input name: Tyler
Tyler is cool!
tkb13@login-teer-28  ~/git/getting-started $ 
```

```c
#include <stdlib.h>
#include <stdio.h>

/**
 * Basic welcome program - prompts for a name, then indicates that person is cool.
 */

int main(int argc, char* argv[]) {
    char name[64]; // buffer for the name, 64 characters long
    printf("Input name: "); // print prompt
    scanf("%64s",name); // read the name
    printf("%s is cool!\n",name); // indicate that the so-named person is, in fact, cool
    return 0;
}
```

Put login.oit.duke.edu to use!

Using your newly configured local editor + remote machine, do the following:

1. Prepare git as described earlier
2. Check out the `getting-started` repository
3. Modify, compile, and run the `welcome.c` program
4. Run `hwtest.py` and confirm your `square.c` is still passing.

ALL DONE?

Nice! Don't head out, though. Work on the current homework and talk to the TAs for help. You can leave if your homework tester shows all passing and you've turned in all the written & code materials.

There are some appendices past here if you're curious about alternative approaches.

APPENDIX: Terminal editors

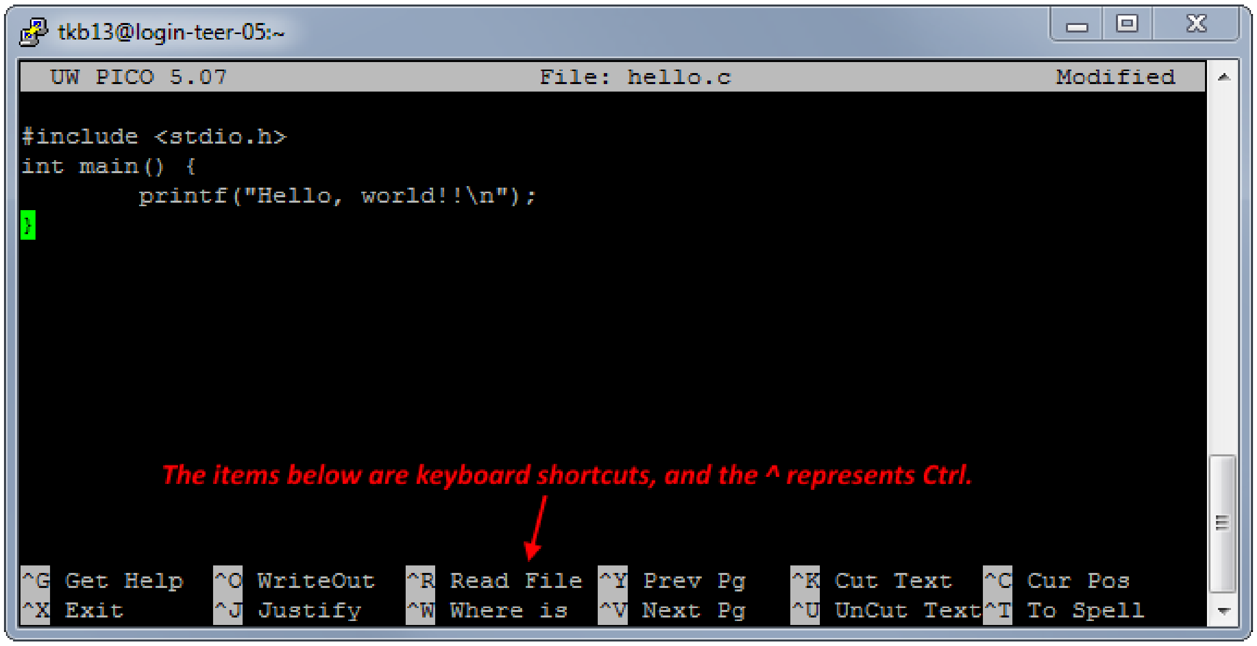
This is optional, but sometimes useful. You can skip it if you want.

In addition to using GUI editors, there are text editors that live entirely in your terminal. The benefit is that you can run them on any machine you can SSH to, so you never have to worry about how to edit remote files with a local tool. The downside is that they're missing some of the nice graphical features GUI editors have. That said, it's useful to have basic proficiency in a pinch, since you don't always have your preferred local editor available in all situations.

Let's test a simple terminal text editor, `nano`. Clone your git repository if you haven't already, then navigate to it. To start editing a file with `nano`, you can type the following at the command line:

```
nano hello.c
```

This line will start `nano` for use in editing a file called `hello.c`. If that file already exists, it will be opened for editing. If it doesn't already exist, a new file with that name will be created and opened for editing.



tkb13@login-teer-05:~
The items below are keyboard shortcuts, and the ^ represents Ctrl.

```
  UW PICO 5.07                    File: hello.c                     Modified

#include <stdio.h>
int main() {
        printf("Hello, world!!\n");
}

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

Once the file is open, modify your hello world program in some way. To save the file and exit the editor, hit Ctrl+X and follow the prompts. Compile and run the program, then use git to commit and push your changes.

I do not recommend `nano` for long-term use in this course – it's a tiny little editor that's good at small quick stuff. For a powerful terminal editor, consider `vim` or `emacs`. Both have a steep learning curve, but people swear by them.

APPENDIX: Synchronization via SFTP

This is optional, but sometimes useful. You can skip it if you want.

Not all remote files are on file servers available via CIFS. In these cases, you can use the SFTP protocol to synchronize local and remote files. SFTP can be used on any host that provides SSH access (such as `login.oit.duke.edu`). This can be achieved by using an editor with a built-in SFTP client (such as Notepad++ on Windows), or using a standalone SFTP client (such as WinSCP) to synchronize local files to the Duke Linux environment. You can also use command-line tools for the purpose, such as `rsync` and `scp`.

Note: The two-factor authentication on login.oit.duke.edu confuses some of these tools, so you might have to play with it.

APPENDIX: Mounting a CIFS share on Ubuntu Linux

This is optional, and only applies if you want to attach your shared home directory from a Linux machine you control, such as a personal computer running Ubuntu Linux or your VCM machine.

Note: You must have root (sudo) access on the Linux machine in question.

First, install the CIFS utilities:

```
sudo apt install cifs-utils
```

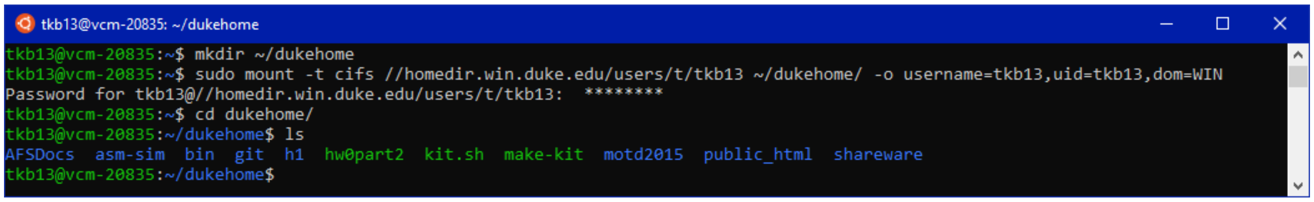
Then, make a directory to serve as the mountpoint. For example:

```
mkdir ~/dukehome
```

The following command will actually mount the network server directory. Here, `<NFIRST>` is the first letter of your NetID, `<NETID>` is your Duke NetID, and `<LOCALID>` is the username on your Linux machine. If on your VCM machine, these will be the same. The command will prompt for your Duke password:

```
sudo mount -t cifs //homedir.win.duke.edu/users/<NFIRST>/<NETID>
   ~/dukehome -o username=<NETID>,uid=<LOCALID>,dom=WIN
```

Note: this is all *one* command. Example execution:



```
tkb13@vcm-20835:~$ mkdir ~/dukehome
tkb13@vcm-20835:~$ sudo mount -t cifs //homedir.win.duke.edu/users/t/tkb13 ~/dukehome/ -o username=tkb13,uid=tkb13,dom=WIN
Password for tkb13@//homedir.win.duke.edu/users/t/tkb13:  ********
tkb13@vcm-20835:~$ cd dukehome/
tkb13@vcm-20835:~/dukehome$ ls
AFSDocs  asm-sim  bin  git  h1  hw0part2  kit.sh  make-kit  motd2015  public_html  shareware
tkb13@vcm-20835:~/dukehome$
```

To later detach the share:

```
sudo umount ~/dukehome
```