

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 become familiar with both 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 the Docker or GradeScope environment is the official testing environment. *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.*

First, we'll get the Duke Docker Container working, then set up Local Tools later on.

For both approaches, we will be using `git` source code control to both back up our data and track its changes.

2. Introducing the Duke Docker Container and git

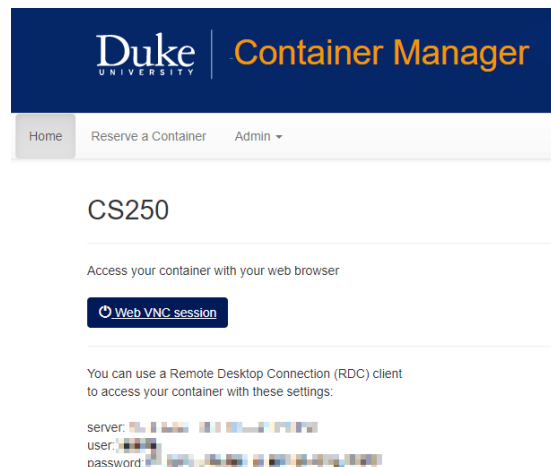
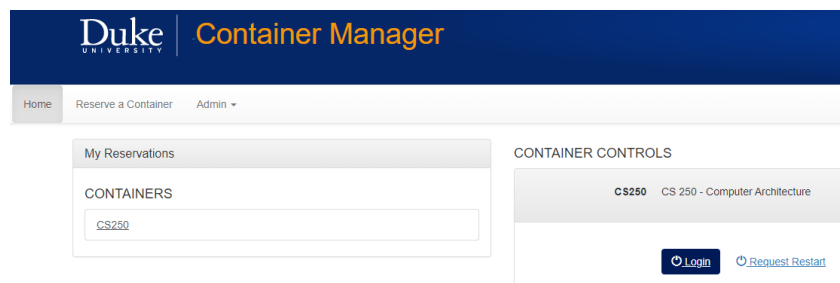
With the help of OIT, ECE/CS 250 provides a container-based online development environment. The containers we provide have all of the programs needed in the class preinstalled.

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.

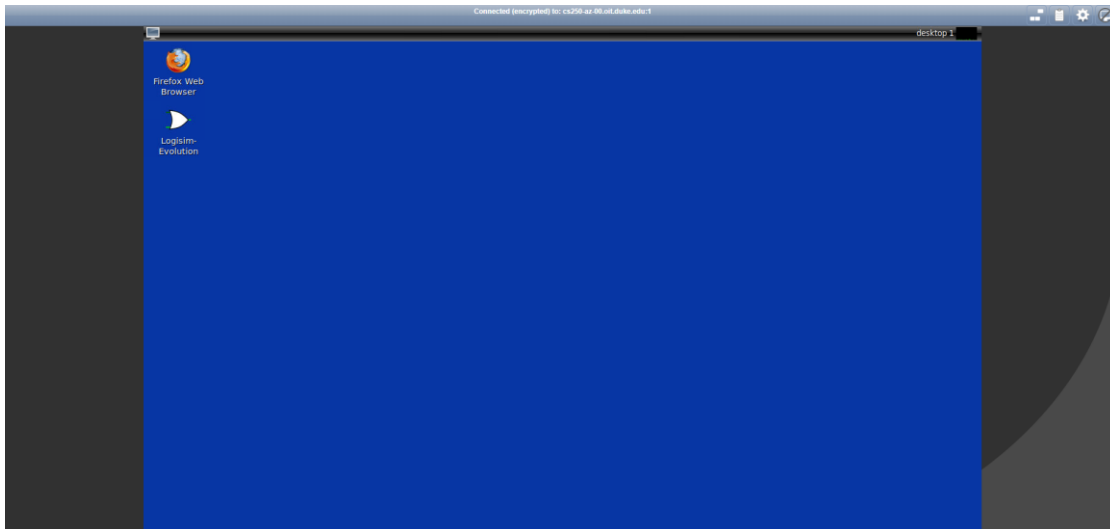
NOTE: File corruption and loss is not a hypothetical scenario, it happens every semester. You should take backing up your work seriously.

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 view connection options. For now, you can connect to your container in your browser with “Web VNC Session”. **If you want a faster/nicer way of connecting, see “APPENDIX: Connecting with Remote Desktop protocol” (HIGHLY RECOMMENDED).** [Connect to your container.](#)




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).

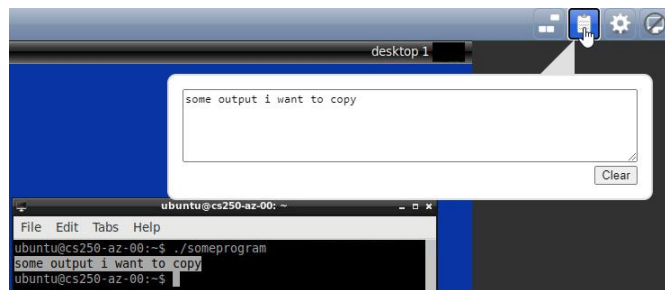
You can drag and drop programs from the start menu onto the task bar at the top to create quick-launch icons.

About the clipboard

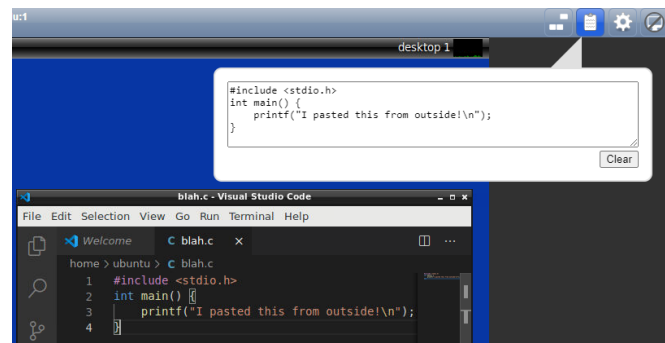
Your docker environment is a separate computing environment from your actual computer, so it has a separate clipboard. This means that copying text on your real computer will not automatically let you paste into your docker environment.

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:



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



Note: *If you think this sucks, you're right! See "APPENDIX: Connecting with Remote Desktop protocol" to connect with a shared clipboard and avoid this issue entirely.*

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

Potential issue (5/12/2022): Does your Visual Studio Code not launch? There's currently a known issue on the container that OIT is working on. To launch it manually, open a terminal and type:

```
code --no-sandbox
```

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; Duke has a deployment of it here: <https://gitlab.oit.duke.edu/>²

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. 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 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 you³. **To make a pair of keys, run the following command in the terminal, replacing “NetID” appropriately:**

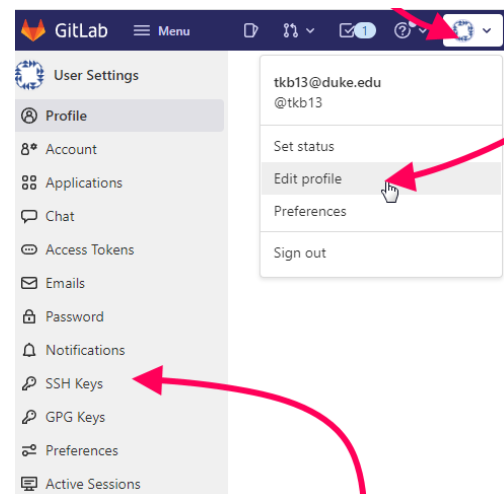
```
ssh-keygen -t rsa -b 4096 -C NetID@duke.edu
```

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://gitlab.oit.duke.edu/> and sign in using the “Duke Shibboleth Login” option. Click on the profile icon in the upper right corner and select “Edit profile”. 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”.

You will need to do the above setup on every system you intend to use git on for this class (including the Docker container and your personal laptop).



² Note: You may find outdated documentation referring to “coursework.cs.duke.edu” – ignore that in favor of gitlab.oit.duke.edu.

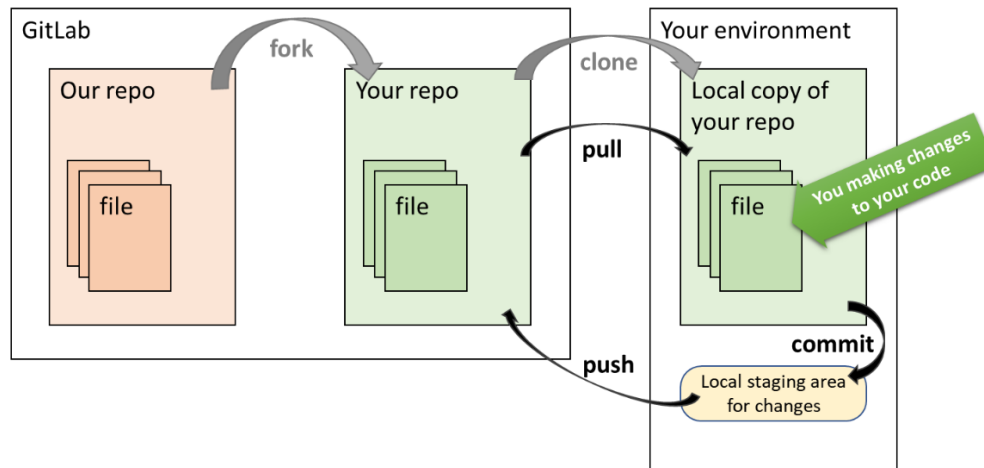
³ You can learn all about this sort of thing by taking an intro computer security class, such as CS 351 or ECE 560.

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 (to GitLab)
- `git pull`: pull changes from upstream (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.

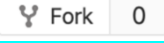


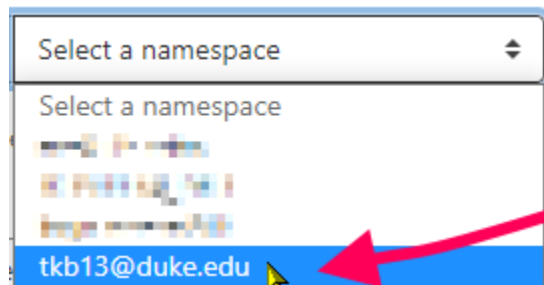
Your First GitLab Assignment

Git step 1: Fork

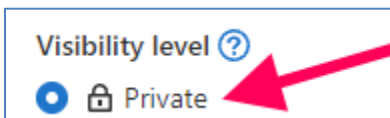
Now let's fork a project (aka repo) from the "Computer Architecture 1Su22" GitLab group, clone the forked project, make changes, commit those changes and push the changes to GitLab. This is the exact same workflow you will use to fork homework assignments, make changes, and backup these changes on GitLab.


Navigate to <https://gitlab.oit.duke.edu/> and log in if needed. Now navigate to Menu > Groups > Your Groups > Kits > Computer Architecture F22. Here you should click on a project titled "Getting Started". If you have trouble finding the project, this [link will take you right there](#).

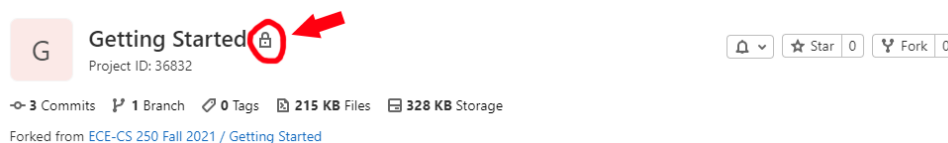
On the upper right click  to fork the project and, under "select a namespace", pick your NetID:



Make it private:



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



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, it 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 directory⁴, 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
```

⁴ 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 `g++` 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



NOTE: There is a connection between this part of the recitation and stuff you need to submit for Homework 1. You may want to give it a read before proceeding.

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:

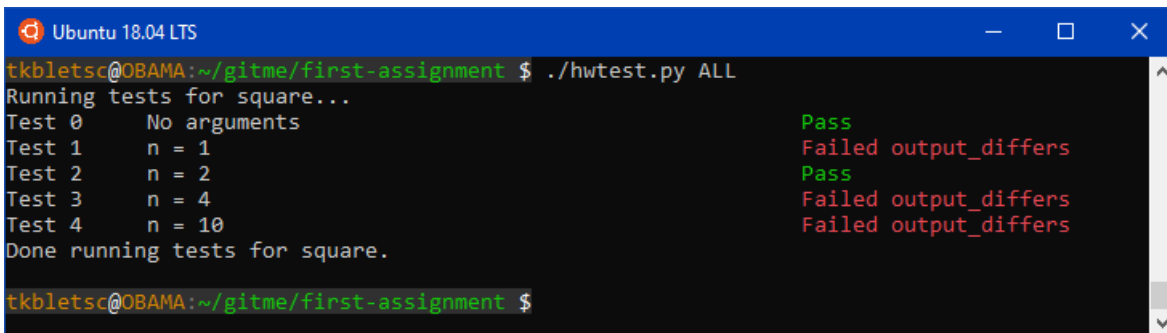
```
usage: hwtest.py [-h] [-C] [-v] [-t TESTDIR] <SUITE_NAME>

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:



```
tkblets@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.

tkblets@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`. **Submit your code to GradeScope** by uploading the `square.c` file). You should get 5/5 points.

This submission will constitute part of your Homework 0 grade.

6. Set up your local computer

We just used the container, which works, but is kind of clunky to use. Let’s get your local computer set up to do assignments as well, then you have your choice of which to use.

The way you get appropriate local tools depends on your operating system. We’ve provided basic directions for Windows, Mac, and Linux, but **you may 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 20.04. If you’re running Windows 10 or 11, 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, however, that we haven’t been able to get the debugging tool `valgrind` working on current Macs. This tool is very useful, so if you get to that point in debugging where you’re chasing segmentation faults and `gdb` isn’t enough, consider doing that kind of debugging in the container so you can use `valgrind`.

Proceed to the “Put that local computer to use!” section.

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.

Proceed to the “Put that local computer to use!” section.

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.

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.

~ END ~

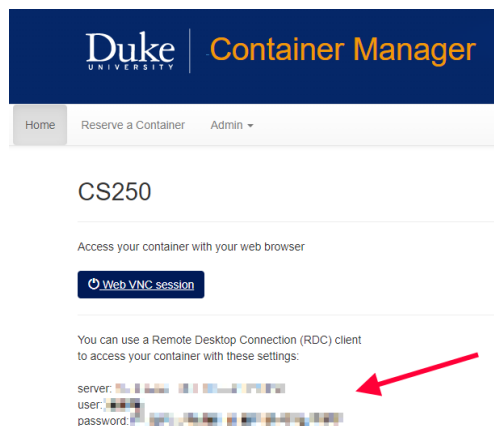
There are some appendices past here if you're curious about alternative approaches.

7. APPENDIX: Connecting with Remote Desktop protocol

The web-based connection to your container is easy, but can be cumbersome:

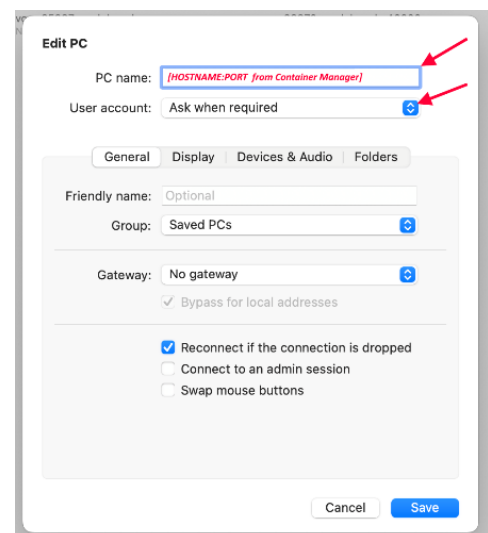
- The clipboard is not shared with your local computer
- The virtual display size of the container doesn't match your local display
- Certain shortcut keys won't work
- Performance isn't great, especially if remote

OIT provides an alternative: you can connect with a **remote desktop client** using the **RDP** protocol. Connection details are given in Container Manager:

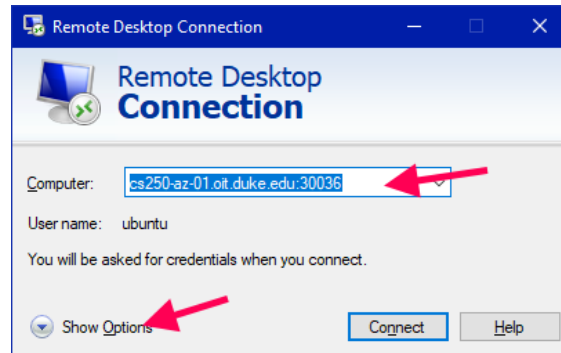


On **Mac**, launch Remote Desktop Connection, and set it as shown on the right. The PC name should be the “server” listed above. Set “user account” to “ask when required” so you can paste in the credentials given. Save and connect, and you should see the same container environment, but in your native resolution with a shared clipboard and better performance.

On **Linux**, install and use a remote desktop client and provide it these settings.



On **Windows**, use the pre-installed remote desktop client. Paste in the server, then hit the Show Options button:



In the options that appear, put in the username given by Container Manager, then check “Allow me to save credentials”. Hit connect, and paste in the password given by Container Manager.

