# PS00 - Hello, world!

COMP590 - The Little Languages of Unix - Spring 2019

GitHub Classroom Starter URL: https://classroom.github.com/a/Jd3VUUHc

## Purpose

The purpose of this problem set is to ensure your course virtual machine is properly setup and to introduce you to our workflow. COMP590 will use GitHub Classroom to host and backup your coursework to private git repositories. Gradescope will be used for autograding submissions with immediate feedback and for hand-grading style checks on your work. You will submit your projects to Gradescope directly from your private GitHub repositories.

Here's an overview of the typical project workflow:

1. Cloning a project
2. Building and running a project
3. Testing a project
4. Committing and pushing changes to GitHub
5. Submitting to Gradescope

## Getting Started

### 0. Setup GitHub and your Virtual Machine (VM)

You will need a GitHub account. If you have not registered for GitHub yet, go ahead and do so now.

To setup your course virtual machine, please refer to the instructions on the course site's resources section. Your virtual machine will need to be up and running before you can continue on.

Finally, be sure you didn't skip the last section of the virtual machine setup where you associate a public key generated on your VM with your GitHub account. This needs to be done in order to complete Step 1.

### 1. Cloning the Repository

When you start a project using GitHub Classroom, you will follow through a link and a process that establishes a clone of our starter code for the project for you. The cloned project will be private, so only you and the COMP590 team can read and write to it.

Open the "GitHub Classroom Starter URL" at the beginning of this document. Once you follow the link, it will ask you to choose who you are in the course roster. Skip past that selection. You will need to accept the assignment. Once it is cloned, follow the link to your personal repository on GitHub. You will see a link to "Clone or Download" the project. You should see "Clone with SSH". If you do not, then click the link "Use SSH". Copy that link to your clipboard, it should start with `git@github.com:...`

You will need to log into your VM again. For detailed instructions, see the last section of the Virtual Machine Setup Instructions for the course. The gist of it, though, is when you return to work on coursework from your host machine, you will need to `cd` into the directory you created your VM, then run the command `vagrant up` to boot up your VM, and finally run `vagrant ssh` to log into it.

While logged in to your VM in a terminal, issue the following commands and replace the words in <tags> with the values specific to you:

```
cd ~
git clone <paste>
cd ps00-hello-world-<your-github-username>
find *
```

Note that the usual Windows shortcut will not work. To paste on Windows, either use the right click menu or press Shift+Insert.

In the two commands above, you `cd`'ed[1] to your home directory, for which the `~` symbol is an alias of, and then issued a `git clone` command which cloned your personal repository on GitHub into your home directory on the VM. Finally, you changed the working directory of your terminal to be your freshly cloned project and used the `find` command to recursively print each of the files in your current working directory. If you click around your repository on GitHub, you'll see the 1:1 correspondence with the files in this directory.

[1] `cd` is the abbreviation of change directory

## 3. Editing Code in `vim`

In this course you will learn how to use the `vim` text editor. Working in a terminal text editor will, for a few weeks, feel awkward and, at times, frustrating compared to programming IDEs like Eclipse or Visual Studio. By the end of the semester, though, you'll be installing `vim` plugins in Eclipse and Visual Studio so that you can edit your programs more productively.

The `main` function of your program is in the file `src/main.rs`. To open it in `vim` simply run:

```
vim src/main.rs
```

The `vim` editor is *modal* and responds to key presses differently depending on the *mode* you are in. When you first open `vim` it is in *Normal* mode. The bar toward the bottom of the screen indicates this mode to you. In normal mode, `vim` is interpretting a little language of *commands* and *motions* based on your keystrokes. First, you will want to navigate to the line where you would like to insert code. You will learn the language of motions in class soon, however these fundamental commands should get you going:

### Basics of Navigating in Normal Mode

| Key | Command or Motion |
|---|---|
| j | Down |
| k | Up |
| h | Left |
| l | Right |
| i | Change to Insert Mode |

### Returning from Insert Mode to Normal Mode

| Key | Motion |
|---|---|
| Ctrl+[ | Return to Normal Mode (Recommended) |
| Esc | Return to Normal Mode |

The line you will want to add in the `main` function uses the `println!` macro and should print a message containing the words hello and world:

```
println!("Hello, world!");
```

Save your changes and quit vim by going back into *Normal* mode (press Control+[ open bracket or press escape) and then entering the command `:wq` (colon, w, q, all together) which is short for *write* file to disk and *quit* `vim`.

### Writing (Saving) and Quitting in Normal Mode

| Key | Motion |
|---|---|
| :w | Write buffer to disk (Save) |
| :q | Quit `vim` |
| :wq | Write and quit |

## 4. Running and Testing

The easiest way to run a project in Rust is using its package manager Cargo. To run your project, after saving and quitting `vim`, issue the command:

```
cargo run
```

You should see your message printed to the screen. To run the built-in integration tests for the project, in this case the same tests that the autograder uses, try:

```
cargo test
```

Both of the tests should pass. The source code of the tests involve some concepts and external libraries, which you just saw compile, that are non-trivial. However, I'll bet if you took a look at the tests in `vim` you would be able to follow along with the comments. The integration tests are stored in the `tests/integration.rs` file.

## 5. Making & Pushing Commits

Once your tests are passing, you will want to commit your changes to your project's git repository. We will discuss `git` in much more depth, but for now you can think of it as a backup checkpointing program. The following commands will make a new checkpoint, called a *commit*, in your repository and upload them ("push" them) to your private repository backup on GitHub.

```
git add src/main.rs
git commit -m "First commit to my project!"
git push origin master
```

Once you have issued these commands, open your browser to your project on GitHub and notice that the changes you made to the file `src/main.rs` are updated in your repository. You should also see the commit message you added above in the commits history.

## 6. Submitting to Gradescope

Once your work is pushed to GitHub you can submit to the Gradescope autograder. To do so, open Gradescope and find the project "PS00 - Hello World". The first time you attempt to upload your work from GitHub to Gradescope you will need to authorize Gradescope to have access to your GitHub repositories. Once you've done that, you can create a submission by selecting your private repository from the Repository selector and `master` from the branch selector. Once uploaded, the autograder should run with feedback shortly.

Using Gradescope, after the late deadline concludes for each assignment, the COMP590 team will be manually grading your code for style and required elements like the Honor Pledge. Be sure you've updated your honor pledge and indented the `println!` statement in your main function for full credit.

Congrats, you've completed the hello world problem set and now have a working setup!