How to get started on the Portneuf server

Created June 2022 by Paul Bodily (updated August 23, 2026)

Average time for assignment completion: 2-3 hours

The material in this tutorial I first encountered as an undergraduate research assistant in 2007. I wish I had had a tutorial like this at that time! **Knowing the basics of command line, SSH, Linux, Vim, and Git makes you extremely marketable and should be included on your CV/resume.** The tutorial is designed with sections be completed in order, but also as a quick reference should you need to come back later. I hope to have created here a resource you can draw back on throughout the semester and beyond!

Important note: All commands on Linux are case-sensitive. Be sure to pay close attention to letter casing (i.e., upper- and lower-case) in all of the below instructions, including the usernames, passwords, folder, and filenames you create. You will risk getting a 0 on the assignment or having to do it all over if your casing doesn't match what is in this tutorial!

If you experience trouble as you work through these steps, get help! Use Google, post on Discord, visit the TA during office hours, ask questions in class, or visit with Dr. Bodily!

Important note: Attention to detail is critical on this assignment. To save you time and to avoid running into excessive problems, please **follow each step in order carefully**. If you have not already done so, I recommend making sure that your operating system (OS) is configured to display file extensions (if you need help, query Google for "displaying file extensions on [your OS]").

Table of Contents

Getting started on a server

Setting Up ISU VPN Access

The Portneuf server is on ISU's private network. This means that you generally have to be on campus connected to the campus network to be able to log into the Portneuf server. To allow you to log into the server remotely, you can first connect to the virtual private network (or VPN). Once connected to the VPN, it is as though you are connected to the campus network and you can then log into Portneuf via ssh with the username, domain (portneuf.cose.isu.edu), and password I sent to you via email. As the use of VPNs to access private networks remotely is a common practice in our field, this, too, will be a valuable learning experience. We have requested that ISU associate VPN access privileges with your ISU credentials, but you need to download a VPN client.

1. For Mac and Windows users: follow the instructions for "Installing the VPN client software" and "Using the VPN client" found here: https://tigertracks.isu.edu/TDClient/1950/Portal/KB/Article/152962/Virtual-Private-Network-VPN-FAQ.
2. For Linux users only: you'll need to search online for a GlobalProtect package for your Linux distribution, e.g., https://github.com/yuezk/GlobalProtect-openconnect. You will then open a terminal window in which you connect to the VPN. E.g.,

   user:$ globalprotect
   >> connect -portal bengalgateway.isu.edu

   Again, your ISU credentials are the VPN access credentials. Then, without closing this terminal, open a second terminal window by which you connect to the Portneuf server.
3. If you have trouble with the GlobalProtect VPN, I recommend connecting with the CoSE IT experts. You can email them at cshelp@isu.edu. You might need to check that your drivers are up to date, uninstalling other VPN clients running on your machine, and/or reinstalling GlobalProtect.

Logging into the server

4. First, make sure you are logged into the VPN.
5. For Windows users running Windows 10 or 11, you will use **OpenSSH** (installed and available by default) to connect to the server. For older versions of Windows, you need to download and install and use **PuTTY** (https://www.chiark.greenend.org.uk/~sgtatham/putty/). For Mac and Linux users, you will use the **terminal** application (installed and available by default) to connect to the server.
6. You should have received a username and password for the Portneuf server. **If running MacOS or Linux**, if my username were `bodipaul`, I would log in by running the following command:

   ```
   ssh bodipaul@portneuf.cose.isu.edu
   ```

   When prompted, enter your password. You're in!

If running Windows, if my username were `bodipaul`, I would open PuTTY, and in the Host Name field I would type:

```
portneuf.cose.isu.edu
```

then hit “Open”. It should then open the terminal window and prompt you for your username and password.

7. You might next see a response like the following (PuTTY users may not):

```
The authenticity of host 'portneuf.cose.isu.edu' can't be
established.
ECDSA key fingerprint is l4UB/neBad9tvkgJf1QZWxheQmR59WgrgzEimCG6kZY.
Are you sure you want to continue connecting (yes/no)?
```

Enter `yes`. You will see a response like:

```
Warning: Permanently added 'portneuf.cose.isu.edu' (ECDSA) to the
list of known hosts
```

You will then see a series of login messages and a command prompt. Congratulations, you are logged in to the portneuf server!

8. (Recommended) Note that there are ways to be able to login to a server account *without having to enter your password each time*. **For MacOS and Linux users**, details are here: https://www.thegeekstuff.com/2008/11/3-steps-to-perform-ssh-login-without-password-using-ssh-keygen-ssh-copy-id/. Complete all three steps. Note that where it refers to "remote-host" you should put in your your username@portneuf.cose.isu.edu (e.g., bodipaul@portneuf.cose.isu.edu).

For Windows users, there are methods to being able to not have to enter your password each time you login, but I haven't been able to find a simple one. I suggest just entering your password each time you log in. (If you find a good one, share it on the class discord!)

9. (Recommended) **For MacOS and Linux users**: You can also *simplify server login* by storing the login details (i.e., hostname, username) in your user-specific SSH configuration file on your machine (you cannot store your password this way, however, as storing passwords anywhere in plain text is not secure, but if you did the previous step, you shouldn't need the password anyway). Details are here: https://linuxize.com/post/using-the-ssh-config-file/.

For Windows users: if you saved your PuTTY session above, you can just double click on the saved session in PuTTY each time you login.

Some basic Linux/Unix terminal commands

When you first login to a server, all commands you enter are by default executed from your home directory. Some terminal commands are a single word (executed by hitting enter). Other commands take additional arguments delimited and separated from the command name using white space. (There are some fun easter eggs hidden in terminals, too.) These commands are ones you will use over and over again, so try them each now and get use to them:

10. `pwd` - this command (which stands for "print working directory") echoes (i.e., prints) back to you the path to the directory you are currently in (referred to as the working directory), which on a freshly opened terminal is the home directory.
11. `ls` - this command (which stands for "listing") echoes back to you a list of the files and folders in your working directory. If nothing is printed, it is because the working directory is empty.
12. `mkdir temp` - the command `mkdir` creates a new directory named `temp`. If you run `ls` again you will see the directory you just created.
13. `cd temp` - the command `cd` (which stands for "change directory") changes your working directory to the directory named `temp`. You can check this by running `pwd`. If you run `ls` again you will see listed the contents of this directory (which, since it is newly created, is nothing).
14. `ls ..` - (note that is a lowercase LS, not a `1`) for any directory, "`..`" is a reference to the directory's **parent directory**. In this case, the `ls` command lists the contents of the parent directory (which if you've been following along will be the contents of the home directory again).
15. `cd ..` - What does this do? You guessed it: changes the current working directory to the current working directory's parent directory.
16. `rm temp` - the command `rm` (which stands for "remove") **is an extremely dangerous command in the terminal—*so be careful with it!*** In this case you will get an error warning because terminals do not let you remove directories without an additional flag. But in general, the `rm` command removes whatever files are listed after the command—and they never come back. There is no "trash" you can pull them back out of—no "undo" feature. **So be very careful.**
17. This time type `rm -r tem` (this is not a typo—leave the 'p' off), but before you hit enter, hit the `tab` key instead. You should see the line automatically completed for you—terminals have tab completion. As long as what you have typed so far has a unique match to a file or folder in your working directory, it will tab-complete (if there are multiple matches, it will list all the matches it finds so far and wait for you to add more letters). Tab-complete will save you a lot of time and headaches!
18. `rm -r temp` - the `-r` is what we call a **flag**. It's a boolean switch that, in this case for this command, tells the `rm` command to execute *recursively*, removing not only the directory, but also recursively removing all of the contents of the directory and any subdirectories.
19. `man rm` - this command (which stands for "manual") shows a manual entry for the command provided as an argument (in this case `rm`). Using the arrow keys, you can scroll up and down and see what types of flags and arguments can be given to this command and what they do. `man` is your friend in learning to use terminal commands effectively (you can also always use Google). When you're done, use the `esc` key to get back to the command line.
20. Another common command is the `mv` command. See if you can use `man` to learn what `mv` does and how to use it. *Hint: it is used both for renaming files/directories and for moving them.*

Git and GitHub

Setting up and adding to your Git repository

Git is a tool that facilitates version control. Using Git you create **repositories** (or 'repos' for short) in which you will put your files and code. When connected subsequently to GitHub it serves as a backup of all of the work you've done.

21. Make sure you are in your home directory (i.e., when you type `pwd`, you should be in `/home/your_user_name`). If you aren't there, then typing `cd` with no arguments should take you there.
22. Create a directory in your home directory called `CS_1337`
23. Change directories to be in the directory `CS_1337` that you just created.
24. To initialize a git repository in this directory, run the

    ```
    git init
    ```

    command. You have now initialized a git repo in the directory `CS_1337`. This is currently just a local repository. Soon we'll connect it to the cloud.
25. The first file we're going to add to the repo is a README file. The next section below will show you how to create and edit files on the command line using a program called Vim. Using these instructions create a file in Vi called README.md using the following command (note that 'md' is short for 'markdown' which allows for us later to put more than just plain text in our README file):

    ```
    vi README.md
    ```

26. Following the instructions on Editing files in the terminal using vi, enter insert mode and type in the following (or something similar—you can always change it later):

    ```
    # Your_name's CS 1337 GitHub Repo

    This repo contains all of the work and assignments completed for CS
    1337, Intro to Computing Systems, taught by Dr. Paul Bodily at Idaho
    State University.
    ```

    Save the file and quit Vi.
27. Once you've added or modified files or folders in a folder containing a git repo, git will notice that the file exists inside the repo. But, git won't track the file unless you explicitly tell it to. Git only saves/manages changes to files that it tracks, so we'll need to send a command to confirm that yes, we want git to track our new file. Use the `git status` command to see which files git knows exist. You will see your `README.md` file listed under "Untracked files". What this basically says is, "Hey, we noticed you created a new file called `README.md`, but unless you use the 'git add' command we aren't going to do anything with it."
28. Add `README.md` to your git repo with the command

```
git add README.md
```

This command adds the file to a "staging environment"; it keeps track of all of the files whose changes *will* be **committed** when next you command git to commit changes (Surprise! Git records changes only when explicitly told to). Use the `git status` command again and you'll see that `README.md` is staged to be committed. **Note:** providing the add command with a directory name instead of a filename will recursively add all files in that directory. Be careful with this—you generally may not want to add everything to your repo!

29. When you commit, git stores a username and email with the commit so that it is known who is responsible for the changes. To set this username and email for all future commits, execute the two following commands with the strings in quotation marks replaced with your information:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

30. Time for your first commit! Run the command `git commit -m "Added README file"`. Every commit requires a message be added. **This message is more important than you realize**. The message serves as a reminder later on about what precisely differentiates this version of your repo from the last version. Without the message, you literally have to sift through the changes to try and remember. Nightmare! Practice now at leaving good commit messages!
31. Use the `git status` command again and you'll see that there is now "nothing to commit".
32. **IMPORTANT FYI**: One should really use branching when using Git. By default, all of your changes will be made on the **main** or master branch. In industrial-scale applications, the main branch is the production version of the application. Say you want to make a new feature but are worried about making changes to the main project while developing the feature. In this case, Git has functionality to allow you to create a new *branch* from one of the versions in the main branch. After you're confident that the branched version works as desired, this can then be **merged** into the main branch. You will invariably learn this in future CS classes. For the sake of keeping things simple, because you will be working solo in this class, and because we're not concerned about any production version code—we will not do branching. But you should be aware that it exists and why it exists.

Editing and creating files in the terminal using Vi

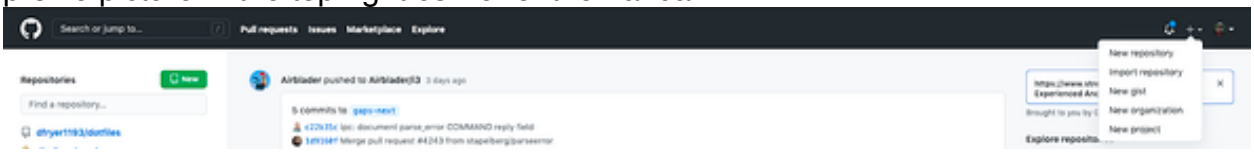
Vi is the universal text editor of Linux. You can use it to edit text files on Linux systems. **Vim** is simply an improved version of Vi, but unlike Vi, Vim is not universal. An excellent resource for learning vim is simply to enter the `vimtutor` command in Linux and follow the tutorial.

33. Once in Vi, you can use the arrow keys to navigate to where you want to make changes.
34. Type '`i`' (for insert) to enter **insert mode**. Once in insert mode you can add and delete characters as normal.
35. Once edits are done, press `esc` to exit insert mode and to return to **command mode**.
36. In command mode, '`u`' is undo.
37. In command mode, type '`:wq`' (':' is preparatory to issuing commands, `w` means "write" (i.e., save), and `q` means "quit").
38. You might consider finding a good Vi or Vim shortcut sheet that you like.

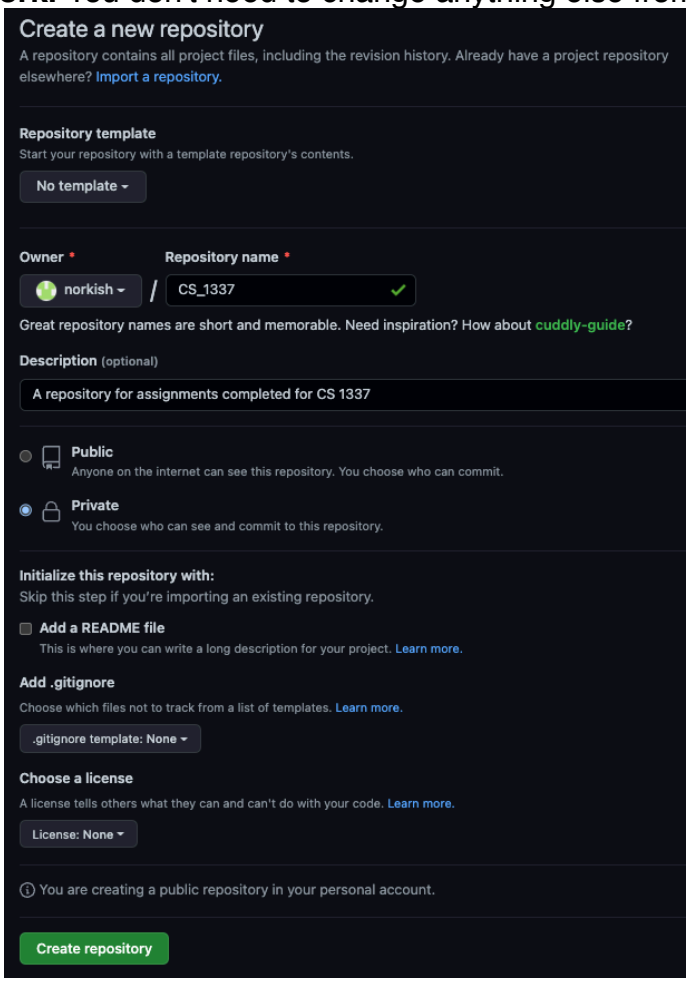
Connecting your Git repo to GitHub

By itself, Git allows you to keep track locally of changes to files in the repository. But the real power of Git is in the ability to back up your repository to the cloud. This is useful for more than just keeping a copy of your data; it comes in handy when you have multiple programmers working on the same project and needing to merge their changes. For this class you'll be working solo on all of your assignments, but all of the commands you'll learn will carry over when you start working on group projects in other classes. **GitHub** is an extremely popular online platform for storing and interfacing with Git repositories. You are welcome to use an existing GitHub account or create a new one for this class.

39. Create and/or login to your GitHub account.
40. On the GitHub home page, find the "New repository" option under the "+" sign next to your profile picture in the top right corner of the navbar:



41. After clicking the button, GitHub will ask you to name your repo (e.g., "CS_1337") and provide a brief description. **You are required for this class to select a 'Private' repo so that your work stays your work.** You don't need to change anything else from the default repo settings.



42. When you're done filling out the information, press the 'Create repository' button to make your new repo. Follow the '....or push an existing repository from the command line' section, which contains three commands you will execute on the server. We'll walk you through them here.
43. **First**, on the portneuf server, push your existing repository to GitHub with the following command[1]. Be sure to replace the URL with the URL for the GitHub repo you just created. **For record-keeping purposes, we will ask you for your github repo URL** (of the form https://github.com/user_name/repo_name) **in the submission for this assignment**.

    ```
    git remote add origin https://github.com/url/to/repo.git
    ```

    Common bug: It is not uncommon for people to put the wrong URL and then when they try to push to their repo they run into an error. You can test that the URL is correct because it should work when opened in a browser. If your origin URL is wrong, you can change it using the command

    ```
    git remote set-url origin https://github.com/url/to/repo.git
    ```

44. **Second**, by default the main branch in Git is named **master** (to see this execute `git branch`). On Oct 1, 2020, GitHub (not Git) announced that in an effort to abandon non-inclusive language, that GitHub "will use **main** as the default branch, instead of **master**." It is up to you what language you wish to use, but this tutorial will use **main** as the primary branch. To name the primary branch **main**, execute the command:

    ```
    git branch -M main
    ```

45. To be able to push your local repo to GitHub, **GitHub requires authentication**. As of August 2021, this requires the use of **personal access tokens** (**PAT**s) rather than passwords. Follow GitHub's instructions for generating a token (classic, NOT fine-grained) as provided here: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token#creating-a-personal-access-token-classic.

    I would suggest for the Note something like "CS 1337 Portneuf Server". Every push to GitHub will require this token, so I would suggest setting an **expiration date sometime after the end of the semester** unless you want to be regularly generating new tokens. **Under "Select Scopes", select "repo".** Click "Generate token".

    Warning: Treat your tokens like passwords and keep them secret. **You will need this PAT every time you push from the server, so save it somewhere.** If you lose your PAT or you are having trouble, you can always delete the token and create a new one.
46. With your Git repo now configured with a remote GitHub repository, you can push the repo to GitHub with the following command. **When prompted, use your PAT in place of your GitHub password**.

    ```
    git push -u origin main
    ```

[1] Note that git commands can be executed from the directory in which you initialized the git repo or any subdirectory of that directory.

It will not show the password as you type it, but it is registering your keystrokes. Note: It can be a little tricky to paste in the Windows Putty terminal, which you'll need to do for this step when entering your PAT as the password In my experience, tapping the trackpad with two fingers works. I suggest practicing pasting into the terminal before being prompted for the password so you can actually see what is being pasted.

47. **You can now view your GitHub repo online at https://github.com/user_name/repo_name.** Since it is private, only those you share the repo with will be able to see it.
48. We need access to your GitHub repo to be able to check that you are successfully backing up your work to your repo. To provide us with access, from your GitHub repo page (i.e., the url referred to in the previous step), select "Settings" along the top menu bar, then under "Access" in the lefthand menu, select "Collaborators." Click the "Add people" button and add the user "cs1337ta" to your repository.

Everyday Git commands

A common issue with using Git is that the version of the repo on your local system gets out of sync with the version of the repo that is in the cloud or that others are using. For this reason it is important to **always follow a specific order of instructions** in your everyday usage of Git (note that with the `add` and `commit` commands you will need to add additional parameters):

49. Stash your local changes (in case those local changes conflict with the remote version):
    ```
    git stash
    ```
50. Update the branch to the latest code (don't forget to use the PAT as the password)
    ```
    git pull
    ```
51. Merge your local changes into the latest code:
    ```
    git stash apply
    ```
52. Add, commit and push your changes
    ```
    git add [subdirs or files to add or with changes to commit]
    git status
    git commit -m "Helpful message describing updates"
    git push -u origin main
    ```

    Note that 1) adding a directory automatically adds everything in that directory; 2) adding is required for any file with changes since the previous commit; 3) issue these commands from the directory containing subdirectory or files with changes to commit; 4) I assume that `main` is the name of your primary branch, and you may need to change that last command if yours is something different.

In this class, since you're working solo, you should never have to merge local changes into the latest code (because local changes should always **be** the latest code). But **it is a good habit to get into to always stash and pull before you add, commit, and push** so that when you begin working with others that you don't try to push your changes to a version of the code that is more up-to-date than the one you were working from. This can lead to overwriting changes that others have made, and although it can be fixed, it can take a lot of time and is no fun!! ***You should always start a coding session by pulling the latest code and you should always end a coding session with the above sequence of commands.***

Getting started coding on the server

The main purpose of this tutorial has been to set up a common coding environment for all students on a remote server. (A secondary purpose is to teach you SSH and Linux commandline.) Within this coding environment we will be exploring the C language and how it gets compiled into assembly and machine languages.

High-level, low-level, interpreted, and compiled languages

In computer science we talk about **high-level languages** and **low-level languages**. You have previously learned **Python** (a high-level language). In this course you will learn **C** (another high-level language). As the purpose of this course is to crack open the proverbial hood and look inside, we will also look at low-level languages like machine and assembly languages with the express purpose of helping you understand how they work.

Low-level languages are sometimes referred to as **machine languages** or **assembly languages**. **Machine language** is the encoding of instructions in binary (0's and 1's) so that they can be directly executed by the computer. **Assembly language** uses a slightly more readable format to refer to the low level instructions. Loosely speaking, computers can only execute programs written in low-level languages. To be exact, computers can actually only execute programs written in machine language. Thus, programs written in a high-level language (and even those in assembly language) have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages. However, the advantages to high-level languages are enormous.

Advantages of high-level languages (this may be on the midterm)

1. **It is much easier to program in a high-level language**. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct.
2. **High-level languages are portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

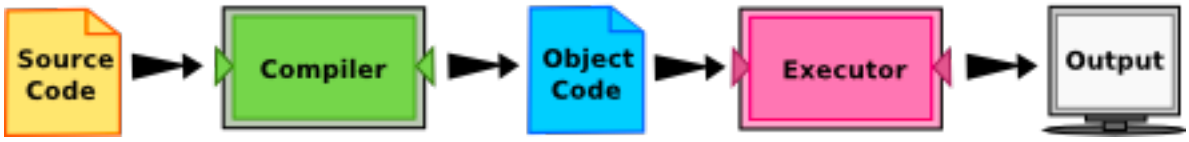
Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for specialized applications (e.g., they're used a lot in some cybersecurity applications), but understanding them will make you a more efficient and effective programmer in general.

Two ways to process high-level programs (this may also be on the midterm)

Two kinds of programs process high-level languages into low-level languages: interpreters and compilers (shown in green below). High-level languages are usually categorized as either **interpreted languages or compiled languages** depending on which process is most commonly used.

1. An **interpreter** reads a high-level program and executes it. It processes the program a little at a time, alternately reading lines and performing computations.

2. A **compiler** reads the program and translates it completely before the program starts executing. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.



Many modern languages use both processes. They are first compiled into a lower level language, called byte code, and then interpreted by a program called a virtual machine. Python uses both processes, but because of the way most programmers interact with it, it is usually considered an **interpreted** language. C is only a **compiled** language.

In this course, we'll essentially start with C and work our way backward to assembly and then to machine code. I hope this excites you. *By the end of this course you will have uncovered the magic and mystery of how a computer works!*

Your first assignment: hello_world.c

53. In your `CS_1337` directory, create a directory called `assignments`
54. Change directories to be in the directory `/home/your_user_name/CS_1337/assignments`
55. Create a directory called `lab_1`
56. Change directories to be in the directory `/home/your_user_name/CS_1337/assignments/lab_1`
57. Each programming language has conventions, i.e., norms that are not mandatory for programs to run, but styles which programmers who use that language generally expect will be followed. Often times those conventions are merely historical, but sometimes they have good practical motivations. **You should generally follow the conventions for the language you are using as regards filenames, commenting, variable and function names, etc.** In the C language, it is convention that **filenames are to be all lowercase with underscores used to separate words**. The suffix for a C source file is ".c". Following this pattern create a file in Vi called hello_world.c using the following command:

    ```
    vi hello_world.c
    ```

58. Following the instructions on Editing files in the terminal using Vi, enter insert mode and type in the Hello World program as found here: https://www.tutorialspoint.com/cprogramming/index.htm. Save the file and quit Vi.

59. Type `ls` to get a directory listing. Do you see your source file?
60. Compile your source code with the command `gcc hello_world.c`
61. Type `ls` to get a directory listing. Do you see your binary object file?
62. Just as "`..`" is a shortcut for a directory's parent directory, "`.`" is a shortcut for a directory itself. Why would we need this? Some commands can be executed from anywhere in the file directory structure (e.g., we just ran `gcc` which is not in the current directory). When we need to distinguish that we're referring to a command or file in the current directory, we use this. Execute your binary object file with the command

    ```
    ./a.out
    ```

    You'll learn later how to specify the name for your object file.
63. Don't forget to add, commit, and push your changes! Navigate back to your CS_1337 directory and let's add your `assignments` directory (and everything in it) to your repository:

    ```
    git add assignments
    ```

    Let's check what got added.

    ```
    git status
    ```

    You should see that it has recursively added your lab_1 directory and its contents. Now we need to commit these changes.

    ```
    git commit -m "Added Lab 1 Hello World in C"
    ```

    Now push to to GitHub!

    ```
    git push -u origin main
    ```

    Again, this last command assumes your primary branch is called `main`. You'll need again to type in your GitHub username and your PAT. You should now be able to see your Lab 1 code in your GitHub repo in a web browser.

A quick GDB tutorial

64. Before we finish, let's see how we use gdb to debug programs in C. You'll first need to navigate back to the lab_1 directory.
    1. To run C programs in gdb requires compiling with the -g flag. Recompile your `hello_world.c` with the -g flag with the command

        ```
        gcc -g hello_world.c
        ```
    2. Next, launch gdb on your binary object file with the command

        ```
        gdb a.out
        ```

3. In gdb, you'll see the program has its own command line. Set a breakpoint at line 5 (in your `hello_world.c` file) with the command

   ```
   break 5
   ```

4. Now run the program with the command `run`.
5. Our `hello_world.c` program doesn't have any variables, so there's not much to look at, but take a minute to visit https://u.osu.edu/cstutorials/2018/09/28/how-to-debug-c-program-using-gdb-in-6-simple-steps/ to see how, in more complex programs, you would print variables. Use this tutorial to figure out how to `step` your way (or `continue`) through the rest of your `hello_world.c` program.

65. For now that's all! You don't need to do anything else. **Nice job!**

A Note About IDEs

66. You are welcome to use an interactive development environment (IDE) on your local machine for editing files for this class if you prefer. Visual Studio Code has a Remote - SSH package that will allow you to connect over SSH. You can then install the C/C++ package on both the host and remote server to have syntax highlighting and error messages. More information on this can be found here and here. You can also complete assignments in a local IDE (e.g., CLion or VSCode) to get help with debugging your C code and then copy the completed code to the server (be sure that it runs on the server before final submission!). All of that said, knowing how to edit files using a remote file editor is an important skill for anyone who spends time working on remote machines. It may not always be practical or possible to configure a local IDE every time you need to modify files on a remote machine.