Hello, World!

This activity is designed to introduce you to using git-keeper, and to compiling and running a C program.

You should have gotten two emails from git-keeper: one with a username and password for the server, and one with a Git clone URL and a link to these instructions.

Git-keeper Assignments

The workflow for most git-keeper assignments will be as follows:

1. Receive an email with a clone URL for a Git repository on the server
2. Clone the repository using the URL from the email
3. Do your work in the local clone of the repository
4. Add and commit your changes to the local repository
5. Push your changes back to the server
6. Check your email to make sure the submission was received and see the results of any tests that were run. If the tests did not pass, you can go back to step 3 and submit again.

Cloning a Git Repository

A Git repository stores the current version of a project's files, along with the history of changes to the repository. When you clone a repository from a server, you get a local copy of the repository on your computer.

Create a directory (also known as a folder) on your computer to store assignments for this class. This guide assumes that you will create a directory on your desktop named `cs110`, and that your username is `username`. You may put the directory wherever you want, but you will have to change the commands below accordingly.

Open up a command line terminal. In macOS, do this by opening the Terminal application. If you are using the Windows Subsystem for Linux (WSL), open up Command Prompt and run `bash`.

Now you need to navigate to the directory that you created for the class. You can do this with the command `cd` (which stands for *change directory*). On macOS you will want to do this:

```
cd Desktop/cs110
```

If you are using WSL, the path to the directory will be a bit longer:

```
cd /mnt/c/Users/username/Desktop/cs110
```

If you are using Cygwin, here is the command to use:

```
cd /cygdrive/c/Users/username/Desktop/cs110
```

When using the command line you are always in a *working directory*. You have now changed your working directory to be your class directory. You can see what your working directory is at any time by running the command `pwd` (print working directory).

Now you can clone the repository for this activity. Copy the clone URL from the email that you received from git-keeper and use it in the command below instead of the URL that is there. In the Windows command prompt you will need to right click to paste.

```
git clone kbhowmik@gitkeeper.wooster.edu:/home/kbhowmik/kbhowmik/cs110/e01-hello_world.git
```

You may be asked if you are sure that you want to connect. You may type `yes`.

You will be prompted for a password. Enter the password you received in the email from git-keeper. For security reasons you will not see any characters as you type the password.

If all goes according to plan you should see output like this:

```
Cloning into 'e01-hello_world'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

The command `ls` lists files and directories. If you run `ls` now you should see a single directory called `e01-hello_world`.

For this class you will run commands on the command line to clone assignments, compile and run programs, and submit assignments. To start, open up the command line interface (open the Terminal application in macOS, open Ubuntu in Windows, or whichever Linux distribution you used for WSL).

Navigating into the Cloned Directory

Run the following command to change your working directory to the directory you just cloned:

```
cd e01-hello_world
```

Now run `ls` and you should see a file named `hello_world.c`.

Note that you can see this file in macOS's Finder or in Windows Explorer too. Open up the directory that you created on the desktop and you should see the `e01-hello_world` folder, and inside there you should see `hello_world.c`.

Editing the File

Launch Atom (or whatever text editor you installed) and open `hello_world.c`.

Note that `hello_world.c` is an empty file. Enter the following into the editor. I recommend typing it out instead of copying and pasting, so that you can start to get C under your fingers:

```
#include <stdio.h>


int main() {
   printf("Hello, world!\n");


   return 0;
}
```

We will talk about what everything in this program means soon, for now you just want to make sure you can edit, compile, and run this program.

Compiling the Program

C programs must be *compiled*. Compiling a program turns it into machine language that is not easily read by humans but can be efficiently executed by a computer.

Compilation is not needed with *interpreted* languages like Python, which can run human readable files directly. This is a trade-off. After making a change to a Python program it can be run again immediately. After making a change to a C program it must be re-compiled before it can be run, but it will likely run more quickly than a Python program that was written to do the same thing.

Compile the hello world program using `gcc` like so:

```
gcc hello_world.c
```

If your program compiled without any errors you should see no output from `gcc`.

Running the Program

Now that the program is compiled, you can run it. By default `gcc` names the compiled program `a.out`. Type `ls` and you should now see 2 files: `a.out` and `hello_world.c`.

You can run `a.out` like this:

```
./a.out
```

You should see the text `Hello, world!` printed in the terminal. Congratulations, you have now compiled and run a program in C!

Naming Your Compiled Program

`a.out` is not a very descriptive name for a program. You can tell `gcc` to name the file something else by using the argument `-o`, followed by the name you want to give the executable. You still need to pass `gcc` the name of the C file to compile too. So to compile `hello_world.c` and name the executable file `hello_world`, you would run the following:

```
gcc hello_world.c -o hello_world
```

The `-o hello_world` portion of the command can also come before the file to compile, like this:

```
gcc -o hello_world hello_world.c
```

Either way, you can now run the program by running `./hello_world`.

Deleting Files

Now that our program is named something better, we don't need the `a.out` file anymore. You can delete it with the `rm` command:

```
rm a.out
```

BE VERY CAREFUL WITH THIS COMMAND!! `rm` does not send files to the trash or recycle bin, it deletes them permanently.

Re-running Commands

You can access commands from the command line history by using the up and down arrows. Try re-compiling the program by pressing the up arrow until you see the `gcc` command that you last used to

compile the program, and then press enter. Then use the up arrow to get back to and re-run the `./hello_world` command.

Configuring Git

When you commit things to a Git repository, Git needs to know your name and your email address.

Set your email address like so (replacing `youremail` with your Wooster username):

```
git config --global user.email youremail@wooster.edu
```

And set your name like so (note that the quotes are required, replace `Your Name` with your name):

```
git config --global user.name "Your Name"
```

You only need to run these two commands once on your computer, and then Git will remember your details in the future.

Committing your Changes

You have edited `hello_world.c`, now you need to add your changes to the Git repository. To do this you must *commit* your change.

You can see what the current status of your Git repository is by running `git status`. Your output should look something like this:

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   hello_world.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    hello_world
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Git notices 2 things: the file hello_world.c has changed since the last commit, and there is a new file called hello_world that is not tracked by Git.

We want the change to the source file hello_world.c to be committed to the repository but we do *not* want to commit the executable file hello_world to the repository. In general you do not want to add executable files to a git repository because they will not work on every computer, they take up more space, and they can be re-compiled on a new computer if need be.

We need to *stage* the change so that it will be included when we commit. To stage a file to be committed, use git add with the name of the file, like this:

```
git add hello_world.c
```

Now run git status again and you should see this:

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   hello_world.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    hello_world
```

where the modification is now green. We are now ready to commit!

Run the following to commit:

```
git commit -m "Modified hello_world.c"
```

The commit message, which follows -m, must be in quotes. If you run git commit without the message you will be taken to the editor vi. If you know how to use vi you can write the message there and save your changes, but vi can be difficult to learn - even saving changes is not straightforward. If you find yourself in vi and you don't know what to do, press escape, type :q! and press enter. That will quit vi without saving your changes.

Once you have committed your change, run `git log`. This will show the history of commits. You should see the initial commit that created the repository, and your commit.

Submitting Your Changes

Now you can push your changes back to git-keeper. Run the following:

```
git push
```

You will be prompted for your password. If your password was accepted and your push succeeded you should see something like this:

```
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 269 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To kbhowmik@gitkeeper.wooster.edu/home/kbhowmik/kbhowmik/cs110/e01-hello_world.git
  a811a69..ef01028  master -> master
```

You may see a warning about the default push behavior not being specified, which you can ignore.

Email From Git-keeper

Now check your email. You should have an email from git-keeper with the results from testing your code. If there were problems with your submission, fix `hello_world.c` and try again by committing and pushing again.

Grading

You will earn up to 2 points for this exercise, broken down as follows:

- 1 point - something was submitted
- 1 point - the tests passed