Hack 1.0

Computer Science I

Department of Computer Science & Engineering
University of Nebraska–Lincoln

Introduction

Hack session activities are small weekly programming assignments intended to get you started on full programming assignments. You may complete the hack on your own, but you are *highly encouraged* to work with another student and form a hack pair. Groups larger than 2 are not allowed. However, you may discuss the problems *at a high level* with other students or groups. You may not share code directly.

If you choose to form a Hack Pair, you *must*:

1. Both join a hack pair on Canvas (go to People then Hack Pairs)
2. You must both work on the hack equally; it must be an equal effort by both partners. Do not undermine your partner's learning opportunity and do not undermine your own by allowing one parter to do all the work.
3. Turn in only one copy of the code under the individual whose last name comes first (with respect to Canvas).

You are graded based on style, documentation, design and correctness. For detail, see the general course rubric.

Category	Point Value
Style	2
Documentation	2
Design	5
Correctness	16
Total	**25**

Table 1: Rubric

For correctness:

- Code itself needs to be correct: 4 pts
- There should be more than one commit: 4 pts
- All commits should have a descriptive comment: 3 pts
- There must be at least 2 contributors: 5 pts

Problem Statement

An essential tool when developing software is a *version control system* (VCS). As you develop software you will make changes, add features, fix bugs, etc. and it is necessary to keep track of your changes and to ensure that your code and other artifacts are backed up and protected by being stored on a reliable server (or multiple servers) instead of just one machine.

A *version control system* allows you to "check-in" or *commit* changes to a code project. It keeps track of all changes and allows you to "branch" a code base into a separate copy so that you can develop features or enhancements in isolation of the main code base (often called the "trunk" in keeping with the tree metaphor). Once a branch is completed (and well-tested and reviewed), it can then be *merged* back into the main trunk and it becomes part of the project.

These systems are not only used for organizational and backup purposes, but are absolutely essential when developing software as part of a team. Each team member can have their own working copy of the project code without interfering with other developer's copies or the main trunk. Only when separate branches have to be merged into the trunk do conflicting changes have to be addressed. Such a system allows multiple developers to work on a very large and complex project in an organized manner.

There are several widely used revision control systems including CVS (Concurrent Versions System), SVN (Apache Subversion), and Git. SVN is a *centralized* system: there is a single server that acts as the main code repository. Individual developers can check out copies and branch copies (which are also stored in the main repository).

Git is a *distributed* VCS meaning that multiple servers/computers act as full repositories. Each copy on each developer's machine *also* contains a complete revision history. This makes git a decentralized system. Code commits are committed to a local repository. Merging a branch into another requires a push/pull request. Decentralizing the system means that anyone's machine can act as a code repository and can lead to wider collaboration and independence since different parties are no longer dependent on one master repository.

Git has become the de facto VCS system in software development. We have provided several external resources below, but this Hack will walk you through the basics of getting started. You will setup a project with git using GitHub (`https://github.com`) as your

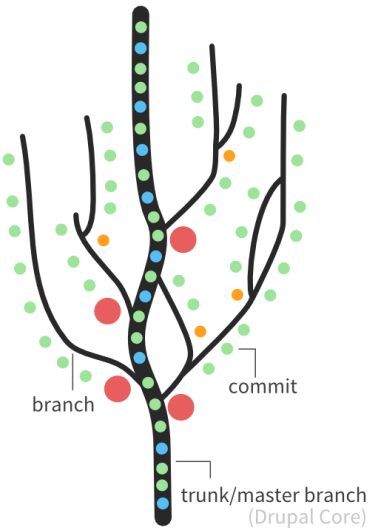


Figure 1: Trunk, branches, and merging visualization of the Drupal project

remote server. You will then collaborate with someone else to commit changes.

This version of the hack assumes that you are in the online section or have chosen to use the CS50 IDE. The following instructions can also be used for *any* command line version of git.

1 Installation

Git is already installed on the CS50 IDE and so no further steps are necessary (though you will need your GitHub account). However, we do need to do some configuration before we start.

Run the following two commands in your CS50 IDE.

```
git config --global user.email "youremail@huskers.unl.edu"
```

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

Where the email and name are substituted with your own. We recommend you use the same email as you used to sign up on GitHub.

2 Creating a Repository

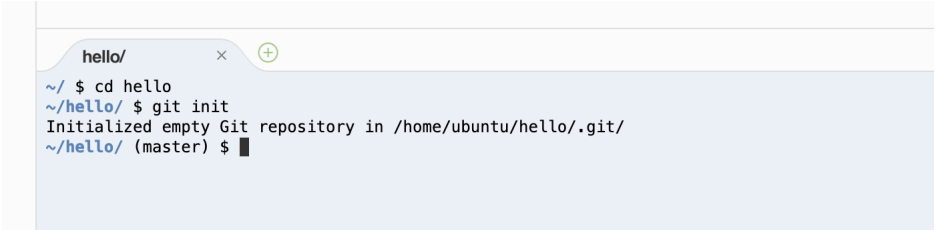
To focus on the git process, you will create and work with a simple "Hello World"-style program but instead of printing "Hello World", it will print your name.

1. Create a new folder (call it `hello`) at the root of your File Navigator and within it, create a `hello.c` source file with code in it that prints your name.
2. Make this folder (and all of its contents) into a git repository by *initializing* it. Before you do, make sure your terminal is in the correct directory (the `hello` directory). Recall the following commands from the previous lab:
   - `pwd` - lists what directory you are currently in
   - `cd hello` - changes your current directory to `hello`
   - `ls` - lists the files in the current directory

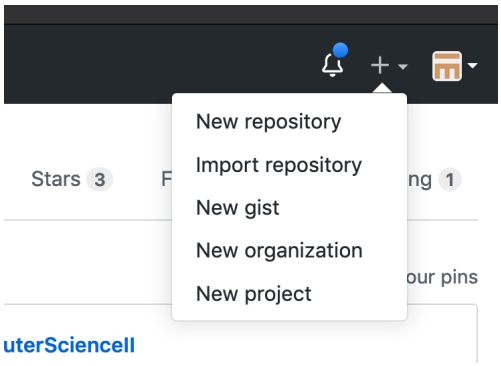
   In the console, change directories to this newly created `hello` directory and execute the following command:

   ```
   git init
   ```

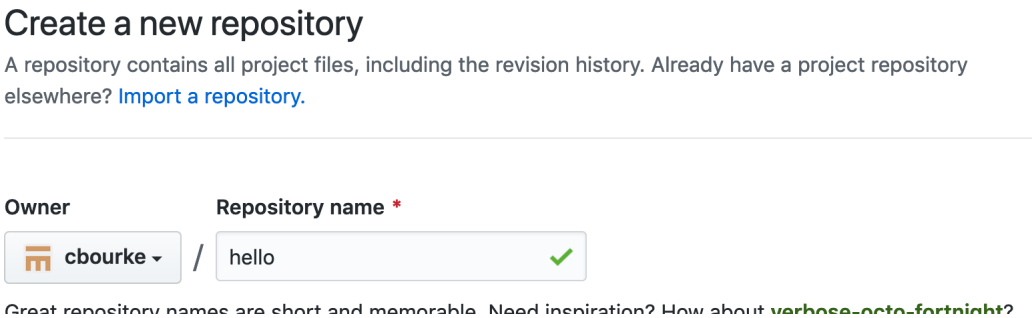
   It will look something like the following.



3. This creates a repository on your CS50 IDE server, but it does *not* create a repository on GitHub. We need to do this separately. Go to your GitHub page (https://github.com/login where `login` is replaced with your GitHub login) and in the upper right, select `New repository`.



4. Name your repo `hello`



5. Go back to your CS50 IDE

6. We now need to make our first *commit* which will commit our code/changes to the repository. You can make as many edits as you want to your source code files and save them, but you only "save" them to your git repo when you *commit* these changes. Enter the following command.

   ```
   git add --all
   ```

   Which will tell git to add all files in your directory (and subdirectories) to the "index" so that changes and updates can be "staged" for the next commit. Don't worry about the jargon for now, essentially this just tells git to add files to its consideration.

7. Enter the following command.

   ```
   git commit -m 'initial commit'
   ```

   This commits your changes to the *local* repository (the repository on the CS50 IDE server). It should look something like the following.

8. We will now *push* these changes to your *remote* GitHub repository. First, we need to specify your GitHub repository as the remote. Execute the following command:

   ```
   git remote add origin https://github.com/login/hello.git
   ```

   where `login` is replaced with your GitHub login.[1]

[1] If you screw this step up and don't replace `login` properly, here's how you fix it:

- Read and follow all instructions going forward
- You can use `git remote -v` to list the remote URL, verify that you screwed it up
- Fix it by running `git remote set-url origin https://github.com/login/hello.git` *change the login to your login.*

9. Now we can push your commit(s) to the remote repository with the following command:

   ```
   git push -u origin master
   ```

   It will likely prompt you for your GitHub username and password.

   Update: as of August 2021, GitHub has changed its security requirements. Instead of providing a password, it now requires you to generate a personal access token. Instructions for how to do this are available here. Be sure to follow these instructions carefully, you need to click "repo" (Full control of private repositories).

   https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-creating-a-personal-access-token

   Enter your GitHub username and your personal access token as your password. If successful, it should look something like the following.[2]

   ```
   ~/hello/ (master) $ git push -u origin master
   Username for 'https://github.com': cbourke
   Password for 'https://cbourke@github.com':
   Counting objects: 3, done.
   Delta compression using up to 16 threads.
   Compressing objects: 100% (2/2), done.
   Writing objects: 100% (3/3), 377 bytes | 377.00 KiB/s, done.
   Total 3 (delta 0), reused 0 (delta 0)
   To https://github.com/cbourke/hello.git
    * [new branch]      master -> master
   Branch 'master' set up to track remote branch 'master' from 'origin'.
   ~/hello/ (master) $
   ```

10. Your repository should now be on GitHub. Point your web browser to https://github.com/login/hello where `login` is replaced with your GitHub user name. You can browse your repository, view its history, etc.

2.1 Git Ignore

Often times there will be files or *artifacts* that you want to create, save and work with in your file system *but* you don't want them committed to the repository. For example, when you compile your `hello.c` program, you probably don't want the executable file, `a.out` to be committed to the repo as you can always rebuild it. The executable file is not part of your source code, but an *artifact* of your code. In general, we want git to *ignore* these artifacts.

To do this, we can create a `.gitignore` file. This is simply a plain text file that contains file and directory names that git will ignore.

1. In your CS50 IDE in the `hello` directory create a new file named `.gitignore`.

[2] Note, you *can* configure git to save your username and password; see https://stackoverflow.com/questions/35942754/how-to-save-username-and-password-in-git-gitextension for instructions. However, since CS50 IDE is a remote server you'd be saving your password in plaintext (very bad idea). That same link has instructions for using an SSH Key instead which is recommended.

2. Edit it and add `a.out` on a single line. Save this file.
3. Commit and push this new file to your local and remote repositories by executing the series of commands:

```
git add .gitignore
git commit -m 'added gitignore file'
git push -u origin master
```

It should look something like the following.

```
~/hello/ (master) $ git add .gitignore
~/hello/ (master) $ git commit -m 'added gitignore file'
[master 89e4035] added gitignore file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
~/hello/ (master) $ git push -u origin master
Username for 'https://github.com': cbourke
Password for 'https://cbourke@github.com':
Counting objects: 3, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 281 bytes | 281.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/cbourke/hello.git
   ca7be09..89e4035  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
~/hello/ (master) $
```

There are many standard `.gitignore` files for various types of projects that you may find useful: https://github.com/github/gitignore

3 Making Changes

You'll often make changes to you code that you should periodically commit to your repository. Though you may make changes and save them to a file, the changes are not saved to the repository's history. Committing is the action that does this. Committing only changes your *local* repository, the changes will still need to be *pushed* to GitHub. In this activity you'll make changes, commit them and then push them to GitHub.

1. Open the `hello.c` file and add a line that prints your major. Also, change the line that prints your name (add an exclamation point at the end or something). Be sure to save your file and compile and run your program to be sure it is correct.
2. Git can automatically track these changes, to display the changes in your console execute the following command.

   `git status`

   Git will display all the changed files that are not yet *staged* (not yet added to the index to be committed). It should look something like the following.

```
~/hello/ (master) $ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

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.c

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

Note that the `a.out` file is not listed (git is ignoring it as we directed it to!).

3. Another useful tool allows you to view the changes or *differences* to a file. Execute the following command.

   ```
   git diff hello.c
   ```

   It should look something like the following. Deletions (or edits) are displayed in red and additions in green.

```
~/hello/ (master) $ git diff hello.c
diff --git a/hello.c b/hello.c
index 3f668b9..621e51c 100644
--- a/hello.c
+++ b/hello.c
@@ -9,6 +9,8 @@

 int main(int argc, char **argv) {

-    printf("Hello World, I'm Chris Bourke!\n");
+    printf("Hello World, I'm Chris Bourke!!\n");
+    printf("My major is Computer Science!\n");
+
     return 0;
 }
```

4. We can now commit and push our changes in the same way as our initial commit. Execute the following series of commands.

   ```
   git add hello.c
   git commit -m 'added my major to the output'
   git push -u origin master
   ```

Some observations/notes:

- In our last commit, we did not use `git add --all` which would have added all (non-ignored) files. Adding everything at once can be convenient but in general commits should only include *related* changes and should be as *fine grained* as possible. You should not use commits as a catch-all/save-all operation.
- You should *always* provide a good, descriptive commit message that accurately reflects your changes. Commit messages provide good documentation on your changes. For example, when fixing a bug, the commit message should reference the original issue or bug report.

4 Collaborating With a Team

In this exercise, you'll need to team up with at least one other person. You'll make them a collaborator on your project so they can make changes and commit/push them to *your* repository on GitHub. Alternatively you can have them make a *pull request*, but these instructions do not cover that; refer to one of the resources in the Additional Resources section for how how to make push/pull requests.

1. On the GitHub webpage, click `Settings` in your project.
2. In the left menu, click `Manage access`
3. Click `Invite a collaborator` and type in your partner's GitHub user name and click `Add`

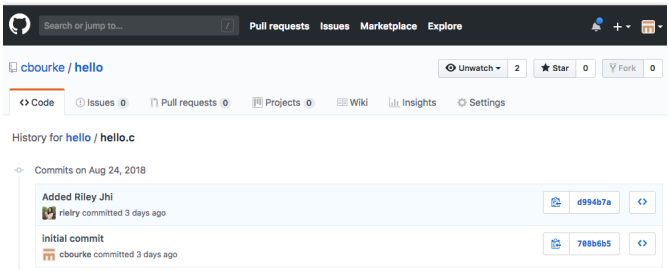
4.1 What your collaborator needs to do

Together with your partner, walk through the following steps. These steps should be done on *their* computer.

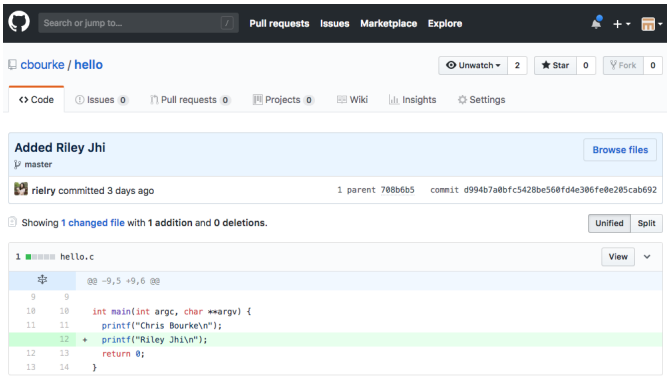
1. Once you've sent an invite to collaborate, they need to accept it.
2. Your partner should create a new directory at the root of their CS50 IDE (or whatever they are using) to hold *your* repository.
3. Your partner should execute the following command inside of the new directory:

   ```
   git clone https://github.com/login/hello
   ```

   Where again `login` is replaced with your GitHub login. This clones the *remote* repository and now you have a *local* version of it!
4. Your partner should add 2 lines of code to print their name and their major.
5. Your partner should follow the same procedure to commit and push their changes to *your remote* repository using the same procedure as they did with theirs.
6. Verify their changes by refreshing your repository on GitHub. You can click on the `hello.c` and if you both did everything correctly, you'll see multiple commits by multiple people:



If you click on `History` you can see the changes for each commit:



Now, go back to *your* CS50 IDE. Remember, your partner's changes were *pushed* to your *remote* repository hosted on GitHub. If you look at your `hello.c` file you won't see their changes because this is your *local* repository.

In order to get your partner's changes you'll need to *pull* their changes from your remote repository to your local repository. To do this, run the following command:

```
git pull
```

As long as there are no conflicting changes, all of your partner's changes should now be reflected in your local repository. Of course, you'll need to repeat this process for your partner's repository (add your name/major to their and commit it so they can pull your changes).

You are *highly encouraged* to start using git/GitHub (or something similar) for all of your future assignments but be sure to commit code to a *private* repository so that you do not violate the department's academic integrity policy.

4.2 Finishing Up

1. Put *your* GitHub URL into a plain text file named `readme.md`. Turn this file in using webhandin. Each individual student will need to hand in their own copy and will receive their own individual grade.
2. Verify what you handed in by running the webgrader which will display the contents of your file.

Additional Instructions

- You are encouraged to collaborate with any number of students before, during, and after your scheduled hack session.
- Each student is responsible for *their* repository, so if/when you team up with a

partner, you'll need to go through this Hack at least twice: once as the primary repository owner and once as a collaborator.

Additional Resources

- Video tutorial on Github Desktop: https://www.youtube.com/watch?v=kFix7UDJ7LA
- Interactive git tutorial: https://try.github.io/levels/1/challenges/1
- Pro Git, free online book: https://git-scm.com/book/en/v2