CSCI 104 - Spring 2017 Data Structures and Object Oriented Design

Lab 01 – Introduction to Git

Git is a distributed source code version control system. When you place your code under version control, you record the changes you make to your files over time and you can recall the history of each of your file changes at will. We will be using git extensively this semester in homework assignments.

GitHub is a development ecosystem based around git. In this course, we will be using GitHub to host our git repositories and we will take advantage of other GitHub features such as the issue tracker and wiki.

Disclaimer

The instructions and discussion in this lab are for the git **command line** interface. GitHub and other software vendors have GUI-based applications to interact with repository. These tools are not will not be supported in this class.

1. Install Course VM

Download and Install the Course VM. Instructions are available here.

2. Git Configuration – SSH Keys

One of the main features of using a distributed version control system such as git, is having a complete backup of your code and its history. Git uses the Secure Shell protocol (SSH) when contacting remote servers. To facilitate this communication, you need to generate a pair of encryption keys: one public and the other private.

In this step, we will generate the set of keys required to use SSH. This will be

done manually through the command line. To start the configuration you would need to launch:

- MacOS or Linux: Terminal
- Windows: Git Bash

Then, type the following command replacing Tommy Trojan's email with your own:

```
ssh-keygen -t rsa -b 2048 -C "ttrojan@usc.edu"
```

When prompted for a file location, use the default location [*press Enter*]:

```
# Creates a new ssh key, using the provided email as a label
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa):
```

After that, you will be prompted for a passphrase to secure your private key. It is a good idea to secure your key with a passphrase, though it is optional.

Upon success, you should receive a confirmation such as:

```
Your identification has been saved in /home/student/.ssh/id_rsa.
Your public key has been saved in /home/student/.ssh/id_rsa.pub.
The key fingerprint is:
01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db ttrojan@usc.edu
```

3. Git Configuration – Developer Profile

The next step is to configure your git *profile* in your development environment. This is **important** because your profile information is used to annotate your *contributions* to a repository/project. Although this may not seem like a big deal when you are the only one committing to a repository, it is of the utmost importance once you work on a group project because this

information is the basis used to calculate your contribution to a project.

You only need to do this configuration once and we will be setting-up the following information:

- Name, e.g. `Tommy Trojan`
- Email, e.g. `ttrojan@usc.edu`
- Editor for commit messages, e.g. `vi`, `emacs`, `notepad`
- Git message colors (make it pretty-er)
- Properly handle new lines

Configuration is performed manually through the command line. To start the configuration you would need to launch:

- MacOS or Linux: Terminal
- Windows: Git Bash

Your Personal Information

Please use your actual first and last name when configuring your git `user.name`. For your email, you should use the email address you want to appear in the git commit log. You should not feel obligated to use your `usc.edu` account here unless you want to. Configure your information as follows:

```
#Set your name and email
git config --global user.name "Tommy Trojan"
git config --global user.email "ttrojan@usc.edu"
```

Note: One of the cool features of the GitHub UI is linking your contributions to your GitHub profile. For that to work, you ***MUST*** register and verify your git `user.email` in the GitHub Email Settings. (Hint: you can have multiple emails registered with GitHub and you could control which ones are public and/or private).

Git CLI Interface

By default, git does not color its output. Pretty printing git messages makes it easy read the output and take proper actions. You can enable colors for interactive use of git by:

```
#Let's get pretty colored output!
git config --global color.ui auto
```

Git Message Text Editor

When committing code in git, the system requires a commit message. If a message is not provided via the commandline, git will launch the Operating System's default text editor prompting you for a message. You can customize this action using the following configuration directive:

```
#Pick your editor of choice to edit commit messages (not code)
# Note that you should pick an editor that is installed on your machine
git config --global core.editor emacs
```

Here, git will automatically launch `emacs`. You may want to change that to your editor of choice. Here are some examples (you only need one):

```
# emacs
git config --global core.editor emacs

# Sublime Text
git config --global core.editor "subl -n -w"
```

Let Git Handle New Lines

Operating Systems implement new lines differently. Here you will configure git to automatically normalize the line feed (`LF`) to properly match the platform. Please use the proper configuration command for your OS:

```
#LF Normalization (Linux & MacOS)
git config --global core.autocrlf input
```

Check Your Work

```
#List local git configuration options
git config --list
```

4. Update Your Github Profile

You need to update your profile to include your name and SSH public key. There are also some optional settings you can change such as your avatar (profile picture) and email notifications.

In your Profile Settings:

- Put your **real name** in the name field. This is to ensure the TAs & Sherpas know who you are regardless of your username.
- [Optional] Change your avatar

In your SSH Key Settings:

- Click on Add SSH Key
- Provide a name for the key, such as "CS104 VM Key", "MacBook Key" or "aludra Key"
- Display the contents of your `id_rsa.pub` file by typing the following command at the temrinal prompt.

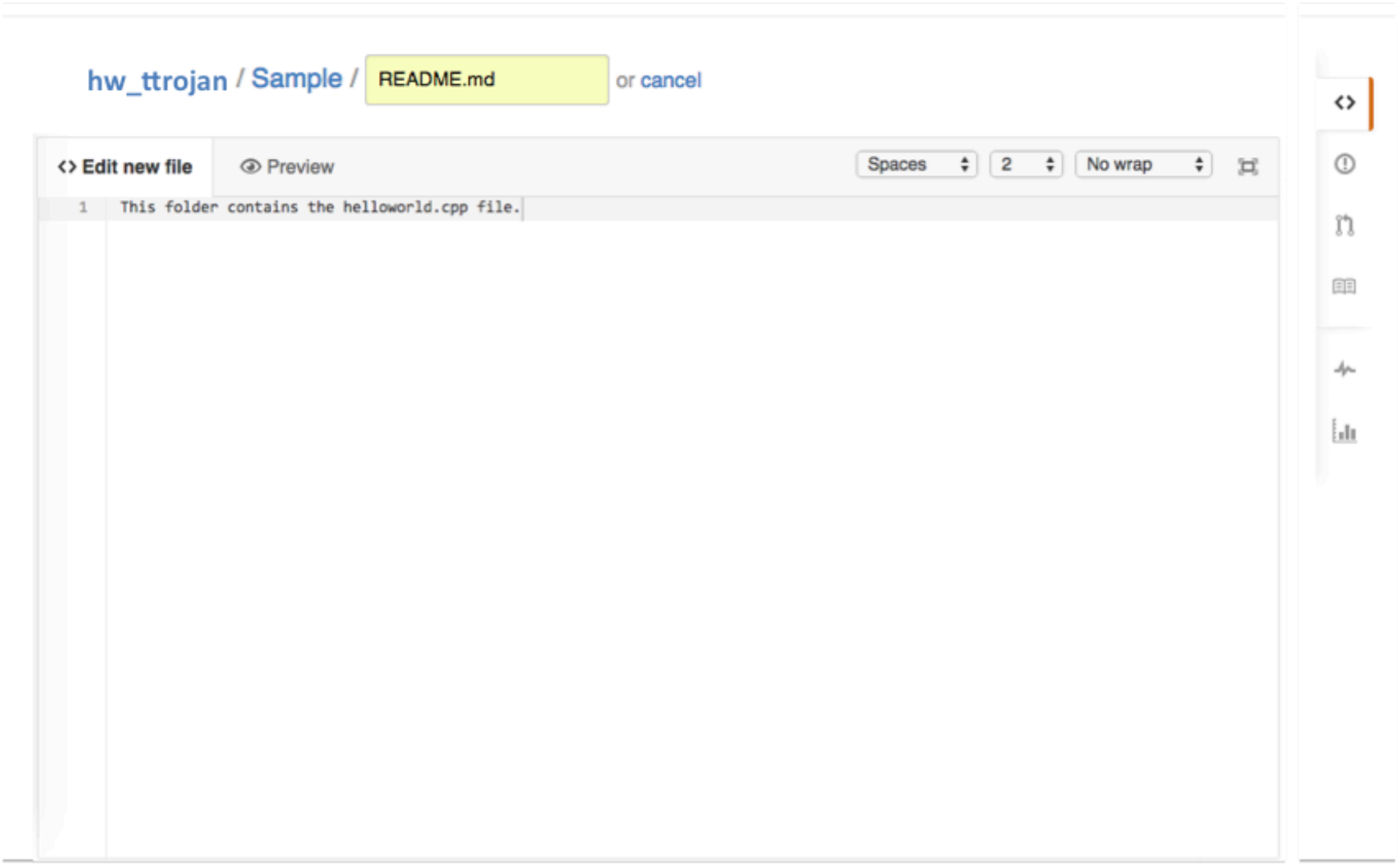
- Copy the contents of your id_rsa.pub file and paste them into the key field. Make sure you copy the entire contents of the id_rsa.pub file. It should start with `ssh-rsa` and end with your email address.
- Click Add Key

[Optional] You can add more emails to your GitHub account using the Email Settings page.

[Optional] Set your notification preferences on the Notification Settings page.

5. Creating Sample Directory

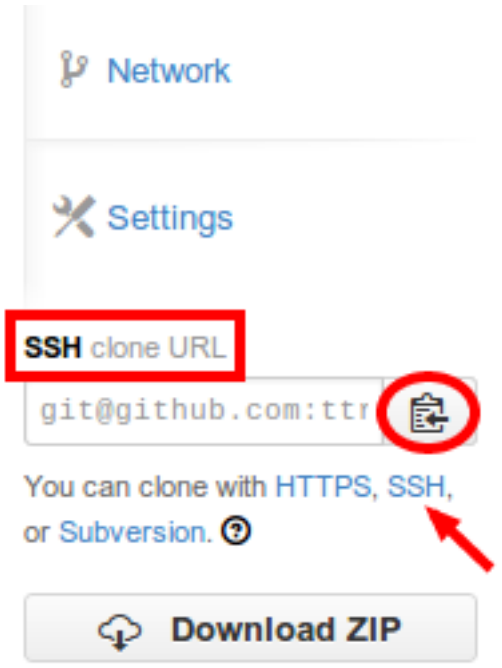
We have created a private repository named `hw-ttrojan` for you (you'll see your USC ID instead of ttrojan). Using GitHub, create a folder named `Sample` inside this repository. First, go to the your `hw_ttrojan` repository and click the "+" (plus) sign and type `Sample/README.md` in the text field that appears. This will create a `README` file inside the `Sample` directory. Readme files are important as these give description of what is containted in the directory. Type a simple description of the sample directory under the `< > Edit new file` tab.



Clone Your Repository

At this stage, we created a place to host our code called a code *repository*. We now need to make a local version of the repository to work with it. The is

called *cloning* the repository. To do that, you need to get the cloning URL by visiting the page of the repository you just created in GitHub and look for the **SSH clone URL**.



Using that URL, you can clone the repo using the following commands in the Terminal or GitBash if you were on Windows.

```
#Change directory to your home directory
cd
#Clone the repository
git clone git@github.com:usc-csci104-spring2017/hw-ttrojan.git
#Start working on the repository
cd hw-ttrojan/Sample
```

HelloWorld.cpp

Use your favorite editor to write a HelloWorld program in a file called `HelloWorld.cpp`:

```
#include <iostream>

int main() {
        std::cout << "HelloWorld!" << std::endl;
}
```

Since we have made progress in our code, it is a good idea to *commit* that code to the repository and make part of our code history. To see what files changed, you run the following command:

You will get something like:

```
On branch master
Your branch is up-to-date with 'origin/master'.

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

        HelloWorld.cpp

nothing added to commit but untracked files present (use "git add" to tra
```

In the above status message, git is telling you that you have one file that is not currently tracked by your repository and that file is called `HelloWorld.cpp`. If you want to add the file to your repository, you need to use the `git add <file>` command. Here is an explanation of the some of the terms:

- **branch master**: you are currently working on the main thread of developement that is called by default: *master*.
- **origin/master**: in this context, origin/master refers to the *master* on GitHub.
- **Your branch is up-to-date with 'origin/master'**: your local repository is sync-ed with the GitHub server. (*Hint*: this is since the last time you received and update from the server).
- **Untracked files**: files that are not part of the repository and were never added to the repository before.

Commiting Changes

To *track* `HelloWorld.cpp`, we use the following command:

```
#Add HelloWorld.cpp
git add HelloWorld.cpp
```

Now, let's check the status of our repository:

You notice we get:

```
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   HelloWorld.cpp
```

This tells us that we just added a new file called `HelloWorld.cpp` and we are ready to commit. Technically, we moved this new file from being *untracked* to be *staged* for commit. You can continue working on other files and you can *stage* them too. Once you feel that you are ready to committed, i.e. be part of the repository history, you use the following command:

```
#Commit with a message
git commit -m "My first HelloWorld using git"
```

This will commit `HelloWorld.cpp` and make it part of repository history. Each commit *must* have a message associated with it. You can add the message as part of the commit command using the `-m` option (see code above) or git will prompt you for a message using the editor you configured in section 3.

To check the commit history for the repository, you use the `log` command:

```
#See the commit history of the repository
git log
```

Which will give us the following message

```
commit df7cd3feda8a856de9cb2dc4bc132f15f7842bb1
Author: Tommy Trojan <ttrojan@usc.edu>
Date:   Tue Jan 14 17:42:52 2014 -0800

    My first HelloWorld using git

commit 6d9fe80012ff9bf5b43120a87dc61bf196fec313
Author: Tommy Trojan <ttrojan@usc.edu>
Date:   Tue Jan 14 15:57:08 2014 -0800

    Initial commit
```

The commit history of this repository shows that we have two commits in reverse chronological order. For each commit, you see the commit id (which is a SHA1 checksum), the author, the time the commit was made and the commit message.

Pushing to the Server

Now that we committed successfully, let's check the status of our repository

```
#Check status of repository
git status
```

You will notice something different about this message:

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

nothing to commit, working directory clean
```

It says that we are: **ahead of 'origin/master' by 1 commit**. If you go back to your repository, you will find that it only has one commit as opposed to the

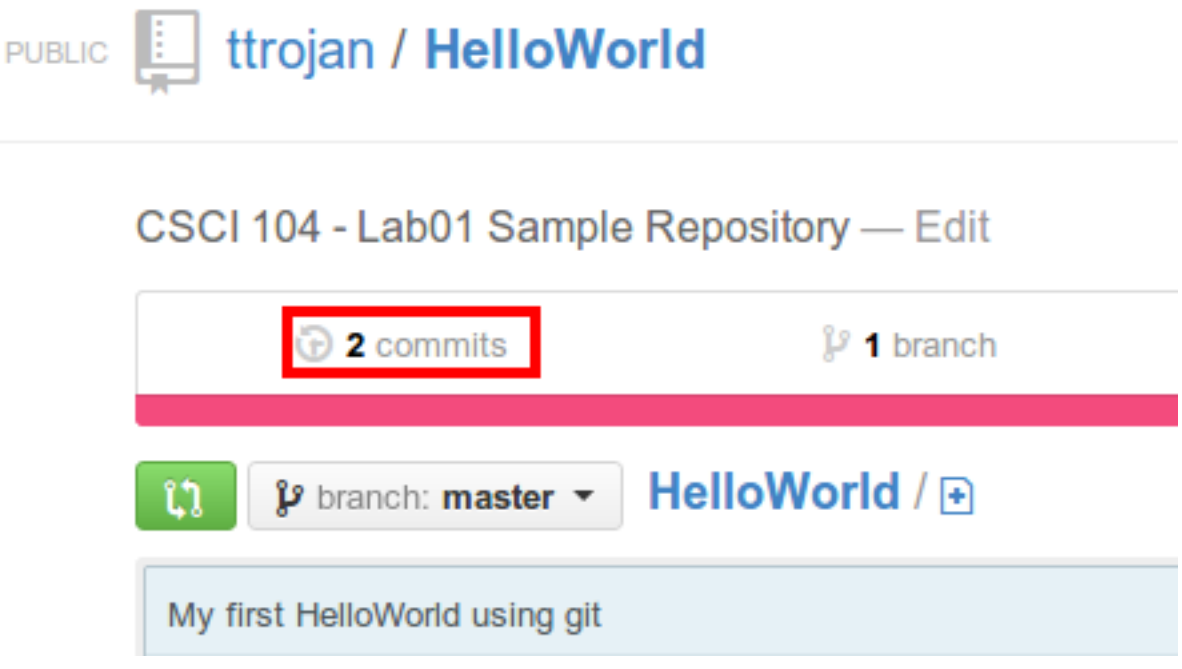
two you have locally. Since git is distributed, it allows you to work "offline" by committing locally and only pushing to the server once you feel ready to do so.

Hint: commit often and commit early. Try to push to the server as soon as you have a good portion of your code complete so you have a backup of your code. Never leave your coding session before pushing to the server.

To *push* your changes to the server, you use the following command

```
#Push changes to remote
git push
```

Now, check the commits section of your code repository by clicking on the *commits* links in your repository's page



Keeping Your Repo Up to Date

Since git uses the distributed model, you can have multiple copies of the repository on multiple machines. This makes it important to make sure your local version is up to date. Here is a practical example of what could happen.

1. Go to your repository on GitHub
2. Click on the `HelloWorld.cpp` file

3. Click on **Edit** to edit the file online
4. Let's update `HelloWorld.cpp`:
    - Change `HelloWorld!` to `FightOn!`
    - In *commit summary* type: `School pride`

Visit the commits page in your repository and note how many commits you have. Now, go to your local repository and check the status of your repository:

```
#Check status of repository
git status
```

You should get

```
On branch master
nothing to commit, working directory clean
```

If you check the commit history, how many commits will you see:

```
#Produce commit history with one line per commit
git log --pretty=oneline
```

To bring your local repository up to date, you need to *pull* from the server:

```
#Get the latest updates
git pull
```

You will get a response that looks like:

```
From git@github.com:usc-csci104-spring2017/hw-ttrojan.git
 * branch            master     -> FETCH_HEAD
Updating df7cd3f..1ffe7e8
Fast-forward
 HelloWorld.cpp | 2 +-
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Here, git is telling you: + It is updating from git@github.com:ttrojan/HelloWorld.git + It updated *master* + The update was from commit `df7cd3f` to commit `1ffe7e8` + Only one file was updated: - File name is: `HelloWorld.cpp` - There was one line *inserted* - There was one line *deleted*

If you want to see in detail what the changes in the *last* commit were, you issue this command:

To see

```
commit 1ffe7e8b5e776395126fb6e06fc72f5af12ab063
Author: Tommy Trojan <ttrojan@usc.edu>
Date:   Tue Jan 14 18:53:08 2014 -0800

    School pride
```

```
diff --git a/HelloWorld.cpp b/HelloWorld.cpp
index 55818b8..b9a284d 100644
--- a/HelloWorld.cpp
+++ b/HelloWorld.cpp
@@ -1,5 +1,5 @@
 #include <iostream>

 int main() {
-       std::cout << "HelloWorld!" << std::endl;
+       std::cout << "FightOn!" << std::endl;
 }
```

Ignoring Files

Code repositories are intended for just that, code. When working on a programming project, you may get a number of different files such as: object files, executables, log files and sometimes compiled header or library files. It is **important** to keep your repository clean of such unecessery files.

git uses a file named `.gitignore` to list all files or file extensions that git should not track or report when you do a `git status`. Your repository now has a list that was auto generated by GitHub. You can see that list by typing the following command from within your repository:

```
#List the contents of .gitignore
cat .gitignore
```

Let's compile our `HelloWorld.cpp` to get the executable `helloworld`:

```
#Compile HelloWorld.cpp
g++ HelloWorld.cpp -o helloworld
#run the executable
./helloworld
```

Now, if we do a `git status`, we will get the following output:

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

        helloworld

nothing added to commit but untracked files present (use "git add" to tra
```

Since `helloworld` is an executable, we don't want to add it to the repository and we definitly don't want `git status` to keep bugging us about it. So, append `.gitignore` to have `helloworld`.

Now, do a `git status` and examine the output:

```
On branch 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 directo
```

```
        modified:   .gitignore

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

As you can see, we are no longer prompted for `helloworld`, however, we now need to commit our changes to .gitignore.

```
#Add .gitignore after modifying it
git add .gitignore
#Commit the changes
git commit -m "added helloworld to the list of ignored files"
#Pushing the changes to the server
git push
```

Let's Git Going

To summarize, we learned the following git commands:

```
#Clone a repository
git clone git@github.com:usc-csci104-spring2017/hw-ttrojan.git
#Add an untracked or modified file
git add <file1> <file2> <...>
#Commit files that are staged for commit
git commit -m "non-optional commit message"
#Push commits to the server
git push

#Check the status of the repo
git status

#Let's make sure our repository is up to date
git pull

#See the commit history
git log
#See the commit history by printing a summary of each commit in a line
git log --pretty=oneline
#See the log of the last commit (-1) showing diffs of all the files (-p)
```

```
git log -p -1
```

Git Resources

For more on git and how to use it, see the Git Resources page.