Git and GitHub

Author: Mike Izbicki

Using Git and Github

In this class we will be using a version control system called git. Version control systems (VCS) are widely used in industry and in open source projects. They are the tool that lets many programmers work together on large, complex software. I don't know what programming language you will use at your future job (it may not even exist yet!), but I guarantee you will be using version control.

In this lab, you will learn the basics of how to use git and Github. Github is a web extension of git that supports online repository and developer's collaboration.

Start with a Simple Tutorial

Start with this tutorial to give you a basic understanding of the git workflow. This will make it easier for you to understand the rest of the lab, and should only take about 10 minutes.

Enabling Git on RedHat

The server that is used for CS100 is a version of RedHat Linux, which is a paid version of linux that has the benefit of being extremely stable. The problem with this version of linux is that to keep it stable they do not update its tools to new version very often. The version of the git tool that is installed is not compatible with the online service github, so you will need to enable an updated version of the tool for your account with the following command in your terminal

```
$ source /opt/rh/devtoolset-2/enable
```

This will allow you to use a newer version of git that is compatible with github. **Please remember to run this command every time you login to hammer!** Alternatively you can add it to your .bashrc.

Deliverables

For Part 1, draw the final git repository (on paper) for the exercise on section 7. Show the drawing to the TA to get checked off, and be prepared for questions about the git commands you used in this lab.

For Next Week

Please make your own git cheat sheet and save it to your computer to show to the TA in the lab next week. It should be a list of ~15 commands and their explanations.

Creating Your First Repo (and Some Simply Unix Commands)

Open a terminal, and cd into the directory you will be doing your cs100 work in. Then create a folder named `firstrepo` and `cd` into it:

```
$ mkdir firstrepo
$ cd firstrepo
```

This folder will be the home of your first git repository. Run the following command to initialize it:

```
$ git init
```

All the information for the git repository is stored in a hidden folder called .git. The folder is hidden because it begins with a "." . By default, the ls command does not display these hidden folders. To display them, you must pass the -a flag. Compare the results of the following two commands:

```
$ ls
$ ls -a
```

Now we are ready to add some files into our repo. Every repo in this class must have a README file. Create the file using the following command:

```
$ touch README.md
```

The touch command is a standard unix command. If the input file does not already exist, touch creates an empty file with that name. If the file does already exist, it updates the file's timestamp to the current time. The ls -l command displays the full information about each file in the current directory. Run the following commands:

```
$ ls -l
$ touch README.md
$ ls -l
```

Notice how the timestamp in the first ls is different than the timestamp in the second ls.

We've created our first file, but git doesn't know about it yet. Run the command:

```
$ git status
```

In the output, there is a section labeled "untracked files." Notice that the README file is in this section. We need to add it into our project using the command:

```
$ git add README.md
```

Now, when we run git status, there is a section labeled "Changes to be committed" with the README file underneath it.

Whenever we finish a task in our repo, we "commit" our changes. This tells git to save the state of the repo so that we can come back to it later if we need to. Commit your changes using the command:

```
$ git commit -m "my first commit”
```

Every commit needs a "commit message" that describes what changes we made in the repo. Writing clear, succinct, informative commit messages is one of the keys to using git effectively. In this case, we passed the -m flag to git, so the commit message was specified in the command line. If we did not pass a flag, then git would have opened the vim editor for us to type a longer commit message. Whether or not you use the -m flag is purely a matter of style, but in my experience, it's usually easier to add the flag.

Let's add some actual code to our project. Create a file main.cpp with the following code:

```
#include <iostream>

int main()
{
    std::cout << "hello git" << std::endl;
    return 1;
}
```

Compile and run the code:

```
$ g++ main.cpp
$ ./a.out
```

Then add it to the repo and commit your changes:

```
$ git add main.cpp
$ git commit -m "added the first code”
```

IMPORTANT: Notice that we only added the main.cpp file to our repo, and did not add a.out. Never add executable or object files to your git repo! Only add source files! Tracking executables uses LOTS of disk space, and makes the repo cluttered and hard to read. If we ever see these files in your git repos, your grade on the assignment will be docked 20%.

Let's make one more commit so we'll have something to play with. Run the command:

```
$ echo "This program prints \"hello git\"" > README.md
```

Remember that echo prints to stdout and the > is used for output redirection. So this command changes the contents of the README file.

The command cat prints the contents of a file to stdout. Verify that your README file has changed using the command:

```
$ cat README.md
```

Now run:

```
$ git commit -m "modified the README”
```

Uh oh!

We got an error message saying: "no changes added to the commit".

Every time you modify a file, if you want that file included in the commit, you must explicitly tell git to add the file again. This is because sometimes programmers want to commit only some of the modified files. We can commit the changes by:

```
$ git add README.md
$ git commit -m "modified the README”
```

Git Repos and Trees

Another important use of version control systems is working with multiple versions of the same project at once. This is VERY useful, and is something you will very likely have to do on a daily basis during your career (and something you will have to do for the assignments).

Every version of our repo is called a "branch." A project can have many branches, and every branch can be completely different than every other branch. List the branches in your current project using the command:

```
$ git branch
```

This should list just a single branch called master. This branch was created for you automatically when you ran the git init command.

One way to think of branches is as a nice label for your commit hashes. Your "master" branch currently points to your commit with the message "modified the README." That's why when we ran git checkout master above, it restored our project to the state of that commit. We could also have used git checkout [hash], if you replaced [hash] with the appropriate hash value. But that's much less convenient. When you use git checkout in the future, you will usually be using it on branch names.

Every time we add a new feature to a project, we create a branch for that feature. Let's create a branch called `userinput` in our project by:

```
$ git branch userinput
```

Verify that our branch was created successfully:

```
$ git branch
```

You should see two branches now. There should be an asterisk next to the master branch. This tells us that master is the currently active branch, and if we commit any new changes, they will be added to the master branch. (That is, master will change to point to whatever your new commit is.)

Switch to our new branch using the command:

```
$ git checkout userinput
```

Now run:

```
$ git branch
```

and verify that the asterisk is next to the `userinput` branch. Since the only thing you did was switch branches, the repo tree looks almost the same. The only difference is the asterisk has moved.

Let's modify our main.cpp file so that it asks the user their name before saying hello:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name;
    std::cout << "What is your name?" << std::endl;
    std::cin >> name;
    std::cout << "Hello " << name << "!" << std::endl;

    return 1;
}
```

We commit our changes to the current working branch the same way we committed them before:

```
$ git add main.cpp
$ git commit -m "added user input”
```

Before this commit, the `userinput` and `master` branches were pointing to the same commit. When you run this command, the `userinput` branch gets updated to point to this new commit.

Let's verify that our changes affected only the `userinput` branch and not the `master` branch. First, checkout the master branch, then cat the main.cpp file, then return to the user input branch.

```
$ git checkout master
$ cat main.cpp
$ git checkout userinput
```

We're not done with this feature yet. Whenever you add a feature, you also have to update the documentation! Properly documenting your code will be a huge part of your grade in this course!

Update the README file with the command:

```
$ echo "This program asks the user for their name, then says hello."
> README.md
```

And add it to the repo:

```
$ git add README.md
$ git commit -m "updated README”
```

The way branches are used out in the real world depends on the company you work for and the product you're building. A typical software engineer might make anywhere from one new branch per week to 5 or more new branches per day.

Fixing a Bug

Wait!

While we were working on our userinput branch, someone reported a bug in our master branch. In particular, the main function in our master branch returns 1, but a successful program should return 0. In UNIX, any return value other than 0 indicates that some sort of error occurred.

To fix this bug, we first checkout our master branch:

```
$ git checkout master
```

Then create a bugfix branch and check it out:

```
$ git branch bugfix
$ git checkout bugfix
```

Here's the tree. Notice that the bugfix branch starts where the master branch was because we switched to master before creating bugfix.

Now we're ready to edit the code. Update the main function to return 0, then commit your changes:

```
$ git add main.cpp
$ git commit -m "fixed the return 1 bug”
```

Since you made the commit on the bugfix branch, your tree splits off in another direction and now looks like this:

Merging Branches

We want our users to get access to the fixed software, so we have to add our bugfix code into the master branch. This process is called "merging."

In this case it is a simple procedure.

First, checkout the `master` branch:

```
$ git checkout master
```

Then run the command:

```
$ git merge bugfix
```

This automatically updates the modified files.

Using branches like this to patch bugs is an extremely common usage pattern. Whether you're developing open source software or working on facebook's user interface, this is the same basic procedure you will follow.

With real bugs on more complicated software, bug fixes won't be quite this easy. They might require editing several different files and many commits. It might take us weeks just to find out what's even causing the bug! By putting our changes in a separate branch, we make it easy to have someone fixing the bug while someone else is adding new features.

Merge Conflicts

Our userinput feature is also ready now. We've tested it and are sure it's working correctly. It's time to merge this feature with the master branch. Run the commands:

```
$ git checkout master
$ git merge userinput
```

Ouch!

We get an error message saying:

```
Auto-merging main.cpp
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

This error is called a "merge conflict" and is one of the hardest concepts for new git users to understand. Why did this happen?

In our bugfix branch above, git automatically merged the main.cpp file for us. It could do this because the main.cpp file in the master branch did not change after we created the bugfix branch. Unfortunately, after we merged the bugfix branch into master, this changed the main.cpp file. Now when git tries to merge our changes from the `userinput` branch, it doesn't know which parts to keep from `userinput`, and which parts to keep from bugfix. We have to tell git how to do this manually.

If you inspect the contents of the main.cpp file, you'll see something like:

```
#include <iostream>
#include <string>

int main()
{
<<<<<<< HEAD
    std::cout << "hello git!" << std::endl;
    return 1;
=======
    std::string name;
    std::cout << "What is your name?" << std::endl;
    std::cin >> name;
    std:::cout << "Hello " << name << "!" << std::endl;

    return 0;
>>>>>>> userinput
}
```

As you can see, the file is divided into several sections. Any line not between the <<<<<<<< and >>>>>>>> lines is common to both versions of main.cpp. The lines between <<<<<<<< HEAD and ======= belong only to the version in the `master` branch. And the lines between ======= and >>>>>>>> userinput belong only to the `userinput` branch.

The key to solving a merge conflict is to edit the lines between <<<<<<< and >>>>>>> to include only the correct information from each branch. In our case, we want the return statement from the `master` branch, and all of the input/output from the `userinput` branch. So we should modify the main.cpp file to be:

```
#include <iostream>
#include <string>

int main()
{
    std::string name;
    std::cout << "What is your name?" << std::endl;
    std::cin >> name;
    std::cout << "Hello " << name << "!" << std::endl;
    return 0;
}
```

Once we have resolved this merge conflict, we can finalize our merge. We first tell git that we've solved the conflict by adding the conflicting files, then we perform a standard commit:

```
$ git add main.cpp
$ git commit -m "solved merge conflict between userinput and master
branches”
```

As you can see, resolving merge conflicts is a tedious process. Most projects try to avoid merge conflicts as much as possible. A simple strategy for doing this is using many small source files rather than a few large files. Of course, in most projects merge conflicts will be inevitable. That's just the reality of working on large projects with many team members.

Exercise

Given the same repo above, draw the tree that results after running the following commands. You will have to submit this to the TA before the end of lab.

```
$ git branch -d userinput
$ git branch -d bugfix
$ echo "everything is awesome" > README.md
$ git add README
$ git commit -m "changed the README”
```

You should check the git cheatsheet to figure out what the `git branch -d` command does.