

# **EE-201**

# **Computer Hardware Skills**

## **Introduction to Version Control Systems and Git**

**Prof. R. B. Darling**  
**Edited by Alvin Cao**  
**Fall 2022**



# Version Control Systems (VCS)

- More than other disciplines, software development runs through a large number of versions before reaching a releasable product, and during the life of the product, many further versions evolve.
- Keeping track of these becomes a nightmare without some system of tracking the changes and versions and revisions.
- Version Control Systems (VCS) have evolved to address this problem.
- A VCS is the most fundamental tool for collaborative software development.
- These are an essential tool not only for software development, but for all types of engineering projects.
- Learning to use these effectively will make you a better engineer.
- On the job, you will be expected to know how to use these.

# Types of VCS

- Centralized (Host-Client)
  - A single repository exists in the cloud
  - Simpler in structure and simpler to operate
  - Changes are available to everyone
- Distributed
  - Repositories exist both locally and in the cloud
  - Faster and can operate off-line
  - Changes can be made available to subsets of users
  - Requires more memory to support the multiple copies
  - Updating a long history can take a lot of time

# Examples of VCS

- Centralized: some are legacy
  - Apache Subversion (SVN)
  - Concurrent Version System (CVS)
  - Perforce Surround SCM (formerly Seapine)
  - Atlassian Bitbucket
  - GitHub (now Microsoft)
  - GitLab
  - Piper (what Google uses)
- Distributed: becoming more popular
  - Git
  - Mercurial (what Meta uses)
  - BitKeeper
  - Bazaar

# Key Features of a VCS

- Changes to a set of files are tracked, generating a history of the development path.
- All of this is bundled into a *repository* or *repo* for short.
- Milestone points are marked by *commits* which capture the state and context of the development at that point.
- Departures from the main development path are also tracked as *branches*.
- Consolidation of departures back into another branch are tracked as *merges*.
- The state and context of any commit can be returned to at any time.

# VCS Uses

- What it is really well suited for:
  - Trying things out
  - Testing and validation
  - Undoing changes
  - Tracking and fixing bugs
  - Collaboration
- What it is not:
  - A file backup system
  - A general-purpose file server
  - A social media platform
  - A panacea for sloppy project management

# Syllabus

We will focus first on Git and then on GitHub.

1. Install locally on your machine
2. Learn the basic commands
3. Practice on some typical situations
4. Learn to connect Git with GitHub
5. Practice editing and sharing
6. Examine workflows and good practices

We will work with the command line. There are other GUI interfaces, but the command line provides more precise control and leads to a better understanding of how Git operates.

# Key Commands in Git

init

add

clone

config

commit

fetch

status

branch

pull

log

merge

push

reflog

rebase

diff

stash

checkout

reset

revert



# Basic Command Line/Terminal navigation

The command line should display your current **working directory**. To “access” files within a specific directory, you will have to either provide the path with reference to the working directory, or navigate to a new working directory.

To change directories:

```
$ cd dir_name
```

```
$ cd dir_name/inner_dir_name
```

If your directory name has a space, you will probably need to use quotes:

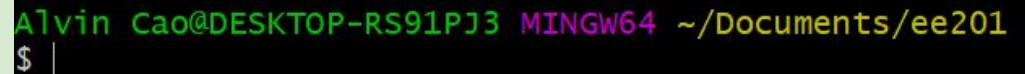
```
$ cd "dir name"
```

To go back to the previous (parent) directory:

```
$ cd ..
```

To list the files and directories in your current directory:

```
$ ls
```



```
Alvin Cao@DESKTOP-RS91PJ3 MINGW64 ~/Documents/ee201  
$ |
```

Useful tip: the command line saves your prior entered commands. To access them, use the up/down arrow keys.

# Setting up a Local Git Repo

## 1. Windows:

- Download and install the latest version of Git: <https://git-scm.com> The latest Windows version is 2.38.0.

Mac:

- Git should be installed by default! To verify, open Terminal and type `git`

## 2. Create a new folder for your repo and open it in (Windows) Explorer/(Mac) Finder.

- ## 3. (Windows) A right click will show both Git GUI Here and Git Bash Here; choose **Git Bash Here** to open the command line. The command prompt is the \$ character. (Mac) Right click -> open Terminal in folder. This opens a command line with this new folder already as the working directory.

## 4. If this is the first time, set up some global configurations:

```
$ git config --global user.name "myname"
$ git config --global user.email myemail@uw.edu
$ git config --global color.ui auto
```

This will set these variables for all future git repos. You can also check the current user.name and user.email by running the above commands without the final name/email parameter.

## 5. Turn the folder into a local Git repo:

```
$ git init
```

This creates the hidden folder `.git` which will contain all of the history and endpoint objects. The directory will now be shown as the branch (master), but the new repo is still empty.

# Setting up a Local Git Repo – .gitignore

1. After initializing a new repo; determine which files you want Git to track. All files in a Git repo are either: (1) tracked, (2) untracked, or (3) ignored. There are a lot of files you may not want Git to track. For example, if you are developing firmware, you will likely want to track only the source code and not the compiled build files for debug or distribution, since those can be regenerated at any time. A .gitignore file can be set up to define what files Git will ignore. First create a new, empty .gitignore file in the root directory:

```
$ touch .gitignore
```

2. Edit this file using Notepad or equivalent to add match patterns of the files to be ignored. For example:

```
# This is a comment. Ignore all files in a directory:  
build/  
# Ignore all files with a specific extension:  
*.log
```

3. Set up and save the .gitignore file BEFORE your first commit.

# Setting up a Local Git Repo – README

1. We will eventually be syncing this local git repo to a remote repo on Github. It's good practice to include a README file which [Github will automatically recognize](#) and use to show a description of your repo.
2. It's standard practice to save your README file as a Markdown file, e.g., README.md - Markdown is a simple syntax for formatting - [see cheatsheet](#) and example below:

README.md
# EE 201 Intro - - - I learned about Git and Arduino in this exercise.

3. Add a README.md file to your local repo (you can use `touch` or just make the file in Notepad)

# Git Reference and Cheat Sheets

- The Git Reference is found online:  
<https://git-scm.com/docs>
- The Visual Git Cheat Sheet is very helpful in learning the key concepts of  
Stash | Workspace | Index | Local Repo | Upstream Repo  
<https://ndpsoftware.com/git-cheatsheet.html>  
Clicking on each of these will show the common commands for moving files between these areas.
- Typing `$ git help <command>` in Git Bash will open a browser window to the full description of that command.
- All Git commands start with “git” to distinguish them from other command line commands.
- Git commands can be run in Command Prompt, Power Shell, or Git Bash. Git Bash provides a more colorful interface that is usually quicker to launch from a given folder.

# Finding Where You Are

## Useful Git commands:

- `$ git status`  
shows the current branch head, any files which have been changed since the last commit, and any files which have been staged for the next commit.
- `$ git log`  
shows the chronological list of commits on the current branch.
- `$ git diff <filename>`  
shows the differences between the last commit and the current file, referred to as 'a' and 'b', respectively.
  - + indicates a line in 'b' which is not in 'a'
  - indicates a line in 'a' which is not in 'b'

# Getting Started With GitHub

- GitHub provides remote repos that local repos can synchronize with.
- To setup a GitHub remote repo, first create an account on GitHub if you do not already have one:  
<http://www.github.com>
- You will need to set up two-factor authorization (2FA).
- Be sure to save your github-recovery-codes.txt.  
These are the only way to regain access to your GitHub repo if you lose your password or 2FA device.
- Once in your account, GitHub offers many items to help learn the system.
- GitHub accounts are free as long as you create only public repos.

# SSH Authentication with GitHub

- Transfers between local Git repos and GitHub require authentication. This can be done with personal tokens through HTTPS, but the most convenient way is to set up SSH authentication.

- If you do not already have a .ssh key, you need to generate one.

Open Git Bash and use

```
$ ssh-keygen -t rsa -b 4096 -C email@uw.edu
```

When prompted for “Enter file in which to save the key”, don’t type anything; just hit enter.

This will by default save the key in a file named id\_rsa.pub.

When prompted for a passphrase, you can also just hit enter to skip.

- This creates private and public keys in your .ssh directory.  
You can optionally create a passphrase to add extra security.

- Start the local ssh agent and add your private key:

```
$ exec ssh-agent bash
```

```
$ eval ssh-agent -s
```

```
$ ssh-add ~/.ssh/id_rsa
```

- If successful, you should see “Identity added: /path/to/id\_rsa ([email@uw.edu](#))” - make note of this path!
- Login to GitHub and within Settings > SSH and GPG Keys -> “New SSH Key”, give it a title (standard practice is to name as machine ID like “Laptop”). In Key, cut and paste your public key, which should be located at ~/.ssh/id\_rsa.pub or the path listed in the previous step.
- With keys on both ends, you should now be able to make transfers.

SSH keys / Add new

Title

Laptop

Key type

Authentication Key

Key

Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'



# Creating a GitHub Repo from a Local Repo

1. Login to your GitHub account.
2. “+” in top right corner -> New Repository, then add a name and description.
3. GitHub will then provide a selection of cut-and-paste text lines that will be needed to link your local repo to the GitHub repo. Quick setup is the fastest. Select the SSH instead of HTTPS option.
4. Copy the link to the clipboard, e.g.  
<git@github.com:myusername/myreponame.git>
5. Go back to your Git Bash window in the local repo.
6. Specify the remote GitHub repo, pasting in the link from GitHub:  

```
$ git remote add origin git@github.com:username/reponame.git
```

This assigns the URL to “origin”.
7. You can verify the current remote address at anytime with  

```
$ git remote -v
```
8. Push the local repo up to the remote:  

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

# Pushing your first commit

1. (optional but good practice) Check the log, which displays the last commit, and the status - you should see your current branch, as well as which local files differ from the last commit:

```
$ git log
```

```
$ git status
```

2. Add all untracked files to the commit index:

```
$ git add -A
```

3. Commit the index with a message describing the commit. In practice this is usually used to briefly describe the changes made (ex. “adding feature A”, “bugfixes for feature B”)

```
$ git commit -m "my commit message"
```

4. Push the commit to master:

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

5. If successful, you should be able to refresh the repo webpage on Github and see your new files show up!

6. When you add/change files locally and want to push a new commit, repeat steps 1-4. In most cases after your first commit you can simplify Step 4 to just:

```
$ git push
```

# Sharing Your Repo With Collaborators

- You can only share access with others who have a GitHub account. Ask your collaborators for their GitHub usernames.
- From the main page of your repo, click on Settings > Manage Access.
- Click on Invite a collaborator and enter their usernames. Each collaborator will receive an email with a link to the repo.
- Once you have added collaborators, you can control what levels of access they may have. For example, you might allow them to pull but not push changes.

# Creating a Local Repo from a GitHub Repo

- Navigate to the top folder that you want to add the new local repo to.
- Right click and Git Bash Here. The command prompt should show the local folder name.
- Clone the GitHub repo, SSH version:  

```
$ git clone git@github.com:username/reponame.git
```

  
or use the HTTPS version:  

```
$ git clone https://github.com/username/reponame.git
```
- This works for public GitHub repos. If the repo is private, you will first need to have been given permission from the owner.

# Key Concept: Branching

- Especially in larger projects, code development rarely follows a single linear path. Subtasks and subprojects often create a local detour from the main flow. So does experimentation.
- Git and GitHub are built on the structure of branches. When you first initialize a local repo, you automatically start out on the branch known as “master”.
- Creating new branches is extremely useful for experimenting with new code or modifications without disturbing the main line that others may be using, or that you yourself may want to later return to.

# Creating New Branches – 1

- Creating a new branch from the current end point is done by, for example: `$ git branch bugfix1`
- To see what branches have been created in a repo, use `$ git branch -v`. A list of all the branches will be shown, and the active branch will be indicated with a `*`.
- Switching to a different branch is accomplished with the checkout command: `$ git checkout bugfix1`.
- Switching branches changes the entire context to that of the selected branch. The working files in the repo will be those associated with the context of the currently active branch.

## Creating New Branches – 2

- **Important!** When you checkout a different branch, any un-committed changes in the current branch will be lost!
- To try out some new code, the best practice is to create a new branch first, starting from a clean working end point. Then checkout the new branch and start making changes within that branch.
- If you then want to go back to the prior end point, commit your work up that point on the new branch, and then checkout the prior branch.
- Git only remembers what has been committed, so bring your work to a good point, perform a commit, and then you will have a clean working end point to branch out from.

# Creating New Branches – 3

- What if you need to create a new branch, but you are not quite ready to commit any changes in the current branch? If you were to checkout a new branch, you would lose those changes.
- Git offers a temporary place to store changes called the stash. To temporarily save your work simply use  
`$ git stash`
- This adds the changes to the top of a stack. To see the current items in the stash use `$ git stash list`
- To apply the changes in the stash top item, use  
`$ git stash pop`
- This allows you to return to your prior working point without using a commit.



# Backing Up and Un-Doing

- To return to the main line from an exploratory branch, just checkout the master branch:

```
$ git checkout master.
```

- Read the Git documentation to learn about other commands such as

```
$ git reset --hard HEAD
```

```
$ git revert
```

```
$ git rebase branchname
```

- These seem similar in name, but they have very different functions. Use the `rebase` command with caution!

# Key Concept: Merging

- Merging allows the changes that have been made to a branch to be blended into the code of another branch.
- For example, if the bugfix1 branch has developed a complete and tested solution, it can be integrated back into the master. First switch over to the branch which will receive the changes

```
$ git checkout master
```

Next, merge in the branch with the changes

```
$ git merge bugfix1
```

- Git will figure out which changes are needed; it is not necessary (or possible) to specify which commits are merged.