CS 370
Revision Control Systems

DR. MICHAEL J. REALE
FALL 2015
Revision Control Systems
- Track revisions/versions of files
  - VERY frequently used to track the current status of code
- Often:
  - Identify WHO made changes to the files
  - Allow creation, deletion, and merging of “branches”
- Also called “Version Control Systems” (or VCS)

Common RCS:
- Git
- SVN
- Mercurial
We’ll be using Git in this course:
- Very popular
- Very powerful

The examples that follow will use Git, but keep in mind:
- The concepts are applicable to other revision control systems...
- BUT
- ...other revision control systems may not work EXACTLY the same way.

By default, Git uses the command line
- However, there are GUI tools to interface with revision control systems (e.g., SourceTree)
Repository

- Contains files that you wish to keep track of
- May use folder hierarchy
Create a Repository

- Go inside the directory that you want to make a repository out of:
  - cd ~/code/MyProject

- git init
  - Creates a new repo out of the current directory
  - NOTE: Files are NOT automatically included in repo!
    - We’ll have to add them to the staging area...
Git Concepts

- There are three “trees” Git maintains:
  - Working directory
    - Contains all files for project
    - NOT all files in this folder are going to have their changes/state saved on commit
  - Index / Staging area
    - List of files whose state you want to save in the next commit
  - Commit tree
    - “Commit” → snapshot/state of files in index / staging area
    - Head → current commit in current branch
Add Files to Repository

- `git add <file>`
  - Add a file to the index/staging area

- `git add <directory>`
  - Add directory and its contents to index/staging area
  - *Example*: `git add .`
    - Adds everything in current directory
Commit

- To save a “snapshot” of the files in the index/staging area, you must **commit** it to the project history

- `git commit`
  - Opens text editor so you can type a “commit message”

- `git commit -m "My message"`
  - Doesn’t open text editor, but instead allows entering commit message on command line
Adding and Committing

- In Git, adding a file to the staging area ONLY applies to the next commit!
  - Once you commit, you have to ADD/RE-ADD any file you want to include in the next commit!
    - You must also add the file AFTER it has been changed!
  - **Advantage:**
    - Allows you to decide which changes you want to group with a given commit (even if other files were changed)
  - **Disadvantage:**
    - Redundant

- **Exception:** `git commit --a`
  - Saves ALL changes made to any files that were ever added to the staging area at any point
Git vs. SVN

- SVN
  - `svn add` → only add files once → tracked until removed from repo
  - `svn commit` → changes committed to central repository (remote repository)
    - `git commit` → only local commits until you push it up to remote repo
  - SVN uses `diff` → tracks changes
    - Git effectively takes “snapshots” → keeps track of contents of whole file
Status of Repository

- `git status`
  - Tells you:
    - Files that are **STAGED**
      - Files changed and included in next commit
    - Files that are **NOT STAGED**
      - Files were changed, but not included in next commit
    - Files that are **UNTRACKED**
      - Files were never included in a commit at any point
  - Good idea to call before a commit to make sure you are saving the right files
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:  main.cpp

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:  readme.txt
    modified:  test.txt

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

    other.txt
If you want to always ignore certain files (i.e., never included in git status), you can add them to a file called `.gitignore`

- One file per line
- Can use star wildcard: `*.txt`
Checking Commit History

- `git log`
  - See ALL commits ever made
  - By default, lists:
    - ID (that is actually a SHA-1 checksum of the commit’s contents)
    - Author
    - Date/Time Committed
    - Commit message
  - **Options:**
    - `git log --stat`
      - Also shows files altered and relative number lines added/deleted from each
    - `git log --oneline`
      - Show each commit on a single line
ASCII Art

- `git log --graph --oneline --decorate --all`
  - Shows an ASCII version of the commit tree
git status vs. git log

- git status
  - Working directory and staging area
- git log
  - Committed history
Tags

- Sometimes you want to mark a certain commit as important
  - Maybe this is “version 1.0” of the software

- Git (and other systems) use **tags** to identify an important commit
Types of Git Tags

- Git has two types of tags:
  - Lightweight
    - Just a pointer to a specific commit
  - Annotated
    - Full objects in Git
    - Contain tagger name, email, date
    - Also can contain tag message
Tag and Release into the Wild...

- **To make a lightweight tag:**
  - `git tag <tag-name>`
    - Tags the current commit with `<tag-name>`

- **To make an annotated tag:**
  - `git tag -a <tag-name> -m “my message”`
    - Note the “-a” option makes it an annotated tag
Tagging a Previous Commit

- To tag a previous commit, just add (the first few characters of) the ID of the commit
  - `git tag v0.5 efof`
Information on Tags

- **git tag**
  - Lists all current tags
  - *Search for tag*: `git tag -l "v1.*"`

- **git show <tag-name>**
  - Shows the commit associated with the particular `<tag-name>`
Suppose we want to look at the previous state of:
- A file?
- The whole project after a certain commit?
- A branch? (We’ll cover this later...)
**git checkout**

- **git checkout master**
  - Return to master “branch” → basically restores project to current state

- **git checkout <commit>**
  - Return to the state after the commit specified by “<commit>”
    - Don’t need to type the whole ID-hash
  - Puts you in **detached HEAD state**

- **git checkout <commit> <file>**
  - Copies old version of <file> from <commit> and overwrites current version of file
  - Adds <file> to staging area

- **git checkout -- <file>**
  - Copies version from last commit (HEAD) and overwrites whatever changes you made after the commit.
  - Adds <file> to staging area
Detached HEAD State

- Usually, HEAD points to the most recent commit on the master branch (or some other branch)

- When using `git checkout <commit>`, in detached HEAD state
  - HEAD points at previous commit
  - Does not mess up repository: master still has most recent version
    - However, you don’t want to make changes without creating a new branch

https://www.atlassian.com/git/tutorials/viewing-old-commits
Checkout Safety

- `git checkout <commit>`
  - Safe; cannot mess up current state of repository

- `git checkout <commit> <file>`
  - DOES modify current state of repository
Changing the Past

- You can effectively “skip” the effects of a commit
  - E.g., you KNOW that a bug was introduced in a given commit, but you may not know exactly where...

- `git revert <commit>`
  - Undoes changes of the given `<commit>`
  - Creates a NEW commit that undoes changes and applies it to the current branch
  - Does NOT remove `<commit>` from history, however
  - Only undoes the effects of a SINGLE commit
git revert vs. git reset

- git revert is comparatively safe

- git reset
  - More versatile, but also potentially VERY dangerous
Resetting Files

- `git reset <file>`
  - Remove a file from the staging area
  - Does not change file’s contents or working directory

- `git reset`
  - Unstage ALL files
  - Again, does not change the files

- `git reset --hard`
  - Unstage ALL files
  - Undo ALL changes since last commit
Resetting Commits

- **git reset** `<commit>`
  - Move current branch tip backward to `<commit>`
  - Reset staging area
  - All changes since that commit are left alone
    - Can be used to combine commits

- **git reset --hard** `<commit>`
  - Move current branch tip backward to `<commit>`
  - DESTROY all changes made after that commit!
    - Both uncommitted AND committed changes!
  - Extremely dangerous!

- You should NEVER reset commits that are already shared / on remote repo!
  - Can reset local commits, however
Branches

- **Branch = separate path of development**
  - Basically your own copy of the working directory, staging area, and commits that goes off on its own tangent
  - Allows you to experiment / work on specific problem without messing with main branch
Branch Commands

- `git branch`
  - Lists all branches
  - At least have “master” branch

- `git branch <branch>`
  - Creates new branch
  - Does NOT checkout new branch (so still on current branch)

- `git branch -d <branch>`
  - Delete branch ONLY if all changes on the branch have been merged

- `git branch -D <branch>`
  - Delete branch, even if it has unmerged changes

- `git branch -m <branch>`
  - Rename current branch
Git branches

- Branches in git = pointers to a commit

- Example:
  - Start with this:
  - Create new branch: `git branch crazy-experiment`
Navigating Branches

- To go to a different branch, we use “git checkout”
  - `git checkout <existing-branch>`
    - Make `<existing-branch>` the current branch
    - Updates working directory to match new current branch
  - `git checkout -b <new-branch>`
    - Create `<new-branch>` and checkout `<new-branch>`
Branches and Detached Heads

- When you checkout, you point to a different commit
  - A “detached head”

- If you DON’T create a new branch, any changes you make effectively have nothing that refers to them

- Don’t make changes on previous commit WITHOUT making a new branch!
Merging

- How do we add the changes we’ve made in our branch back into another branch (like the main branch)?

- `git merge <branch>`
  - Merge `<branch>` into current branch

- `git merge --no-ff <branch>`
  - Same as above, but ALWAYS generate a merge commit
# Start a new feature
`git checkout -b new-feature master`

# Edit some files
`git add <file>`
`git commit -m "Start a feature"`

# Edit some files
`git add <file>`
`git commit -m "Finish a feature"`

# Merge in the new-feature branch
`git checkout master`
`git merge new-feature`
`git branch -d new-feature`
Merging Algorithms

- Two major Git merging algorithms:
  - Fast-forward merge
  - 3-way merge
Fast-Forward Merge

- **Fast-forward merge**
  - Really all git has to do is “fast-forward” down the branch, since there were no other changes in the current branch
  - Not possible if branches have diverged
3-Way Merge

- **3-way merge**
  - If the branches have diverged, need 3 commits:
    - Two branch tips
    - Common ancestor
  - May have to resolve conflicts if both branches modified the same file
Conflict Resolution

- If there’s a conflict:
  - Merge command will stop right before committing and tell you that conflicts are there
  - Use `git status` to see what files have conflicts
  - Modify the file(s)
  - Add the files with `git add`
  - Do a regular `git commit`
So far, everything we’ve done has been on the local machine.

What if you have an existing repo on a remote repository (e.g., on Bitbucket or Github) and want to:

- ...copy/clone repository onto local machine?
- ...push changes on a local repository to a remote repository?
- ...pull changes from a remote repository?
Clone Repository

- `git clone <repo>`
  - `<repo>` = address of the repository
  - Clones repository and creates working copy on local machine
    - Also creates connection to remote address, so that you can pull/push changes to it
    - Connection name is called `origin`
Example: Clone from Bitbucket

- Let’s say I have a project on Bitbucket called AwesomeAndroidProject
  - On Bitbucket there’s a link for the address of the repo:

![Repository Address](image)

- Then, to clone the repo on the local machine:
  - `git clone https://DrMichaelReale@bitbucket.org/DrMichaelReale/awesomeandroidproject.git`
Remote Repositories

- `git remote`
  - Lists remote connections you have to other repositories
  - `git remote --v`
    - Shows URLs as well

- `git remote add <name> <url>`
  - Adds connection with `<name>` and address `<url>`
Push Changes to Remote Repo

- If you make changes to the local repository, you have to manually push those changes to other remote repos

  - `git push <remote> <branch>`
    - Push local commits on branch `<branch>` to `<remote>`
    - NOTE: Will not push commits if results in non-fast-forward merge!

- `git push <remote> --all`
  - Push ALL local commits
NOTE: Tags are not automatically pushed to the remote repo!

- To push a specific tag:
  - `git push origin v1.5`

- To push all tags:
  - `git push origin --tags`
**Fetching Commits from Remote Repo**

- **git fetch <remote>**
  - Fetches branches from remote repository
    - Also downloads any files/data it needs
    - Stored as **remote branches**
  - Does NOT mess with your commits/branches/etc.

- To view remote branches: *git branch --r*

- You can then decide whether to merge any of these branches or not.
Pull Changes from Remote Repo

- If you want to fetch AND merge, then use:
  - `git pull <remote>`
    - Same as:
      - `git fetch <remote>`
      - `git merge origin/<current-branch>`
References

- Atlassian Git Tutorials: https://www.atlassian.com/git/tutorials/setting-up-a-repository/
- Automated Tutorial for Git: https://try.github.io/levels/1/challenges/1