

---

# CSE 374

# Programming Concepts & Tools

Hal Perkins

Spring 2022

Lecture 16 – Version control and git

# Midterm

---

- Maybe a little too long, but it the main purpose was to review/consolidate “what we’ve done” and it did that
- Grading is mostly done
  - Still need to take care of a couple of things, then expect to release scores and sample solution in the next day or two
- More about scores and “what it all means” for course grades on Wed. But basically this is only about a fairly small % of the course grade, and even if it didn’t go well, it’s only part of the picture and is not a make-or-break score.

# Administrivia

---

- HW5 due Thursday night. How's it going?
- HW6 writeup posted soon – please read when available
  - Part I due a week from Thur. – skeleton code & Makefile
  - Full assignment due following Thursday
  - Detailed discussion in next couple of classes
- Partner survey posted before class, due 11 pm Wed.
  - *Please* follow instructions – it's not hard!
  - Will create gitlab repositories for groups after that
    - Watch for email from `gitlab.cs.washington.edu` with repo information and link, then log on, set up keys, clone repo, and try it out

# Administrivia (added Wed.)

---

- HW4 feedback released last night
  - Remember that gradescope’s “total score” is meaningless. We keep track of things like extra credit and program operation vs code quality separately. If you do not do any extra credit parts it does not affect (i.e., lower) your course grade.
- HW5 due tomorrow night. How’s it going?
  - Watch the late days! Some people are down to 1 or 0 (updated numbers in the canvas gradebook). If you run out of time, submit your best effort at the deadline for partial credit. No credit after that.
- HW6 writeup coming soon – please read when available
- HW6 partner survey due 11 pm tonight
  - *Please* follow instructions – it’s not hard!
  - Will create gitlab repositories for groups after that
    - Watch for email from [gitlab.cs.washington.edu](mailto:gitlab.cs.washington.edu) with repo info and link, then log on, set up keys, clone repo, and try it out

# Discussion board request

---

- *Please* use descriptive titles and provide enough context so readers can find relevant postings and understand them without a treasure hunt.
  - Can someone else discover what your posting is about from the title? If not, please try to fix that.
    - Suggestion: Avoid using “clarification” or “question” in titles 😊
      - e.g., “clarification on hw5” is not helpful; “is there a max dictionary size” is much more useful, especially if it in the hw5 category
- And please use public postings for general questions so all can benefit

# Where we are

---

- Learning tools and concepts relevant to multi-file, multi-person, multi-platform, multi-month projects
- Today: Managing source code
  - Reliable backup of hard-to-replace information (i.e., sources)
  - Tools for managing concurrent and potentially conflicting changes from multiple people
  - Ability to retrieve previous versions
- Note: None of this has anything to do with code. Like make, version-control systems are typically not language-specific.
  - Many people use version control systems for everything they do (code, papers, slides, letters, drawings, pictures, . . . )
    - Traditional systems were best at text files (comparing differences, etc.); newer ones work fine with others too

# Version-control systems

---

- There have been many plenty – sccs, rcs, cvs, subversion, git, mercurial, perforce, sourcesafe – but git is the dominate one in use today
- Terminology and commands aren't particularly standard, but once you know one, the others aren't difficult – the basic concepts are the same
- git: distributed version control
  - Every user has a full copy of the repository; allows easy branching & merging for large collaborations (e.g., linux kernel)
  - git is the basic tool. Can be used alone or with centralized repositories. GitHub is the best know; we will use CSE GitLab (almost identical) for hw6

# What is git?

---

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of "It's really pretty simple, just think of branches as..."; and eventually you'll learn the commands that will fix everything.

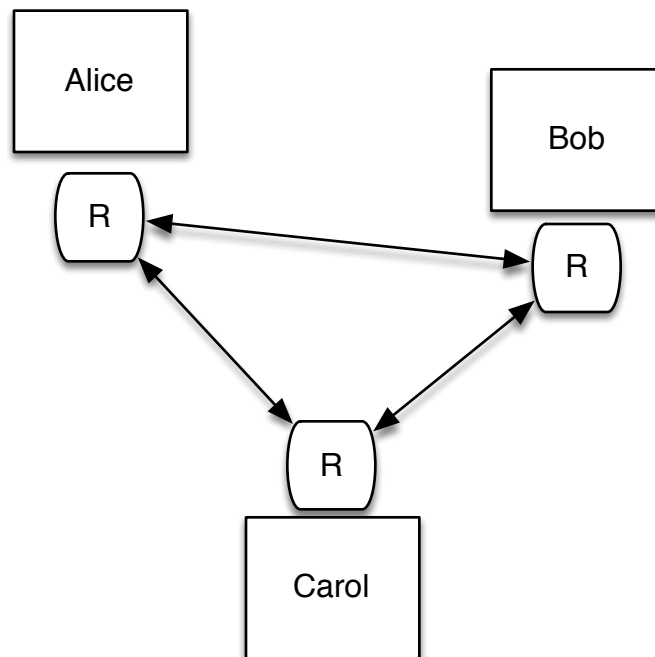




# Git basics – general version

---

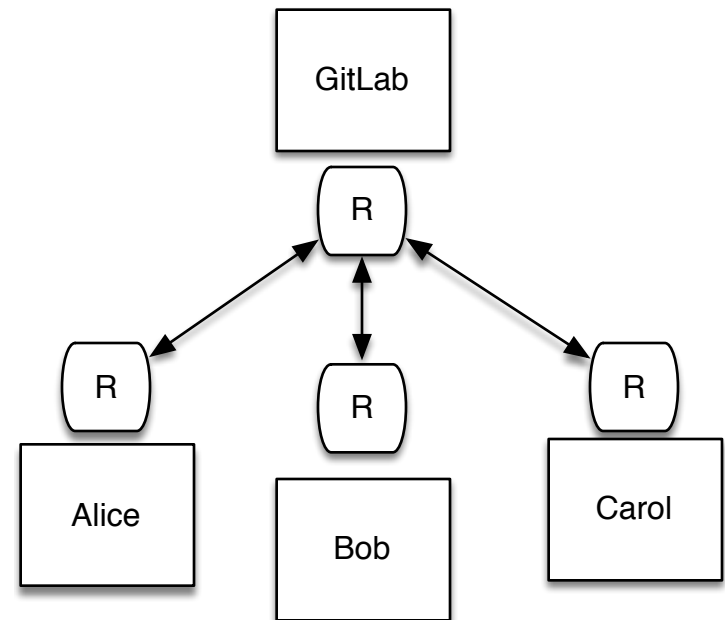
- A project lives in a repository
- Each user has their own copy of the repository
- A user *commits* changes to their copy to save them
- Other users can pull changes from that repository



# Git basics – central repo (we'll use)

---

- Users have a shared repository (called *origin* in the git literature; for cse374 it is your group's repository on the CSE GitLab server)
- Each user *clones* the repository
- Users *commit* changes to their local repository (clone)
- To share changes, *push* them to GitLab after verifying them locally
- Other users *pull* from Gitlab to get changes (instead of from each other)



# Tasks

---

Learn the common cases; look up the uncommon ones.

In a production shop using git...

- Create
  - a new repository/project (rare – once or twice a year)
  - a new branch (days to weeks; not in cse374, but used in production shops for independent development)
  - a new commit (daily or more, each significant change)
- Push to repo
  - regularly, when you want to back up or share work – even with yourself on a different computer
- Other operations as needed (check version history, differences, ...)

# Repository access

---

A repository can be:

- Local: run git commands in repo directory or subdirectory
- Remote: lots of remote protocols supported (ssh, https, ...) depending on repository configuration
  - Specify user-id and machine
  - Usually need git and ssh installed locally
  - Need authentication (use ssh key with GitLab)
- cse374/HW6 use ssh access to remote GitLab server
- Feel free to experiment with private, local repos or private repos on gitlab

# Getting started (GitLab)

---

- Create local ssh keys (ssh-keygen) and add to your GitLab account (instructions on gitlab, linked from cse374 git tutorial; only need to do this once)
- Set up a repository (we'll do this for you on hw6; if you do it yourself you get to pick name, location)
  - +New Project (on gitlab web dashboard)
- Clone a working copy of the repo to your machine
  - cd where-you-want-to-put-it
  - git clone git@gitlab.cs.washington.edu:path/to/repo
    - url for above comes from gitlab page for your project, find using link in email you get when project created or on by logging in to gitlab
    - If git asks for password, keys aren't set up right – fix it!

# Set git Defaults

---

- Git keeps a local `.gitconfig` file in your home directory with info like your name, email, and other settings
- Most defaults are sensible, but there are some that you should set so git won't nag you every time you try to commit or pull
- Enter these commands once before you use git the first time:
  - `git config --global user.name "Your Name Here"`
  - `git config --global user.email netid@uw.edu`
  - `git config --global pull.rebase false`(of course, substitute your own name and email address)

# Routine git/GitLab local use

---

- Edit a file, say `stuff.c`
- Add file(s) to list to be saved in repo on next commit  
`git add stuff.c`
- Commit all added changes  
`git commit -m "reason/summary for commit"`
- Repeat locally until you want to push accumulated commits to GitLab server to share with partner or for backup...

# git/GitLab use (sharing changes)

---

- Good practice – grab any changes on server not yet in local repo before any local commits  
    git pull
  - Also do this any time you want to merge changes pushed by your partner
- Test, make any needed changes, do git add / git commit to get everything cleaned up locally
- When ready, push accumulated changes to server  
    git push
- If push blocks because there are newer changes on server, do a git pull, accept any merge messages, cleanup, add/commit/push again



# File rename/move/delete

---

- Once files have been committed to gitlab repository, need to tell git about changes to git-managed files
  - `git mv files`
  - `git rm files`
  - git will make the changes locally then make corresponding changes to remote GitLab repo when you push
  - If you use regular shell mv/rm commands, git will give you all sorts of interesting messages when you run `git status` and you will have to clean up 😊

# Demo

---



# Some examples

---

- Update local copy to match GitLab copy  
git pull
- Make changes  
git add file.c  
git mv oldfile.c newfile.c  
git rm obsolete.c
- Commit changes to local repo  
git commit -m "fixed bug in getmem"
- Examine changes  
git status (see uncommitted changed files, will also show you how to revert changes, etc.)  
git diff *file* (see uncommitted changes in *file*)  
git log (see history of commits)
- Update GitLab shared repo to reflect local changes  
git push

# Conflicts

---

- This all works great if there is one working copy.
- But if two users make changes to their own local copies, the two versions must be *merged*
  - git will merge automatically when you do a “git pull”
  - Usually successful if different lines or different files changed
- If git can’t automatically merge, you need to fix manually
  - git will tell you which files have conflicts (git status)
  - Look in files, you will see things like

```

<<<<<<<< HEAD
for (int i=0; i<10; i++)
=====
for (int i=0; i<=10; i++)
>>>>>>> master
```
  - Change these lines to what you actually want, then add/commit the changes (and push if you want to)

# git gotchas

---

- Do not forget to add/commit/push files or your group members will be very unhappy
- Keep in the repository *exactly* (and *only*) what you need to build the program
  - Yes: foo.c foo.h Makefile
  - No: foo.o a.out foo.c~
  - You don't want versions of .o files etc.:
    - Replaceable things have no value
    - They change a lot when .c files change a little
    - Developers on other machines can't use them
- A simple .gitignore file can be used to tell git which sorts of files should not be tracked (\*.o, \*~, .DS\_Store (OS X) )
  - Goes in top-level repo directory; useful to push to GitLab and share (we will push a default one in new hw6 repos)

# Summary

---

- Another tool for letting the computer do what it's good at:
  - Much better than manually emailing files, adding dates to filenames, etc.
  - Managing versions, storing the differences
  - Keeping source-code safe
  - Preventing concurrent access, detecting conflicts
- git/Gitlab tutorial for CSE 374 on website
- Links to GitLab on website and in CSE 374 tutorial
  - And in the mail you get when your repo is created
- Full git docs and book are online, free, downloadable
  - Beware of complexity – much of what they describe is beyond what we need for CSE 374. Keep it simple!