Multiple copies of code

- Imagine if every time you wanted to snapshot (or “branch” or share) your project, you made a copy

```
$ ls cs220-eg/
hello_world.c  Makefile  README

$ cp -r cs220-eg cs220-eg-2018-02-01

$ ls cs220-eg cs220-eg-2018-02-01
cs220-eg:
hello_world.c  Makefile  README

cs220-eg-2018-02-01:
hello_world.c  Makefile  README
```

Multiple copies of code

- Suppose you “snapshot” or share your code frequently, and at different times on different machines
  - Up to you to remember “meanings” and relationships of copies
  - Lots of copies = lots of space; waste, redundancy
  - Difficult for team members to track where the latest version of each file lives

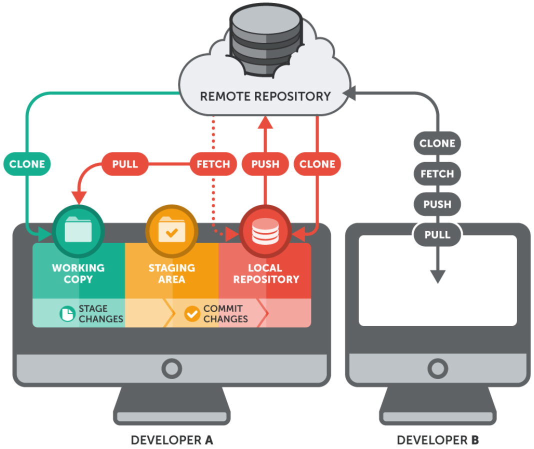
Git to the rescue

- A *repository* (“repo”) stores all versions of all project files, and their entire histories back to the beginning of the project
    - Repos eliminate disadvantages of the “lots of copies” method, while still facilitating snapshotting, branching, sharing
    - Cleverly organized to avoid storing redundant data
    - Repo (master/origin) can be *local* (on your computer) or *remote* (e.g., on bitbucket.org or github.com)

Git

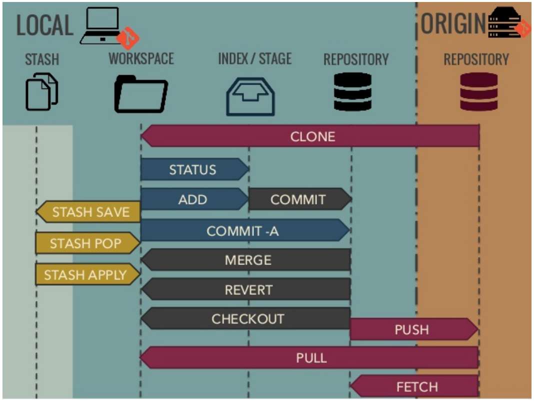
- Git is a way of sharing files; like DropBox or GoogleDrive, only much more powerful (and great for sharing code)
- Distributed version control
- Facilitates collaboration, snapshots, sharing
- Basic software skill, along with programming
- Works with any programming language; really, any project that consists of mostly text files

Git



- From `www.git-tower.com/learn/git/ebook/command-line/remote-repositories/introduction`

Git



- From `www.slideshare.net/origamiaddict/git-get-ready-to-use-it`

Git

- In your working copy, you can go about your usual business:
  - Editing files (with `emacs`, `vim`, etc)
  - Compiling and executing files
- But you'll also perform some repo-related tasks
  - `git add <file>`: add to project ("stage a file")
  - `git commit -m "commit message"`: update local repo to include changes since last commit ("take a local snapshot")
  - `git push`: send changes up to remote repo (on bitbucket)
  - `git status`: check what's been modified or staged, etc.
- Can't modify a repo directly using plain-old `mv` or `rm`; all interactions are via `git` command
  - `git mv <file> <file>`: rename a file
  - `git rm <file>`: remove a file (delete it)

- Full list: `services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf`

Git

- Files that are part of your project (you `git add`'ed them) are called *tracked*
- Tracked files can be in one of a few states
    - Unmodified (same as copy in local repo)
    - Modified (different from copy in local repo but not yet staged)
    - Staged (next `git commit` will update repo)
- editing files: Unmodified -> Modified
- `git add`: Modified -> Staged
- `git commit`: Staged -> Unmodified
- Information about changes in a copy of the repo is stored across several non-human-readable files in a subdirectory called `.git`
    - This subdirectory gets created for you when you clone a repo

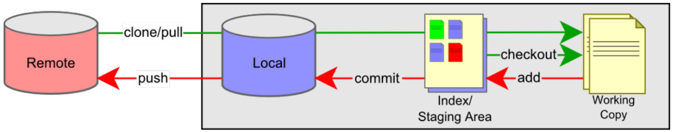
Git

- Files that are *not* yet part of your project ("unstaged") are called *untracked*
  - When you create a new file; it's *unstaged* until you `git add` it
  - But git will notice it, and it will appear as unstaged if you check your `git status`
- Some untracked files are are files that we want git to "ignore", because we'll never want to include them in the remote repo
  - Tell `git` to ignore a file by adding it to `.gitignore` file
  - Good candidates for ignoring might be `a.out`, `gitlog.txt`
  - We'll discuss this again soon

Git

- After `git clone` occurs, syncing between local and remote repos accomplished via `git pull` and `git push`
  - `git pull`: local repo asks for most updated copy from remote repo
  - `git push`: local repo sends all recent *commits* up to remote repo

Git


Remote
clone/pull
push
Local
commit
Index/
Staging Area
checkout
add
Working
Copy

Git

- Workflow Suggestions
  - Start each work session with `git pull`, to ensure your local copy is up-to-date
  - After you complete work on a small task, `commit` it
  - Include a message with every commit to explain what changes you committed (use -m, or you might be forced into an editor to create one!)
  - Make sure you commit and push before the end of each work session
  - To see a record of your latest commits displayed on the screen, you can type `git log`

Git

- Don't be discouraged if `git` concepts are elusive at first
- You can get by with just a few key ideas
- `commit` early, `commit` often
- Tutorials and explanations linked from Resources section of Piazza (go to General Resources area, then click on Tools Reference)
- Lots of help available from CAs, instructors, Google, ...

Git

- Today, we want everyone to have access to class resources for this section
    - our class repository (repo) is hosted by bitbucket.org
    - can view the shared files in a web browser, but we want *local* copies to work with
    - today you'll *clone* the class repo into your ugrad account
    - when instructors add more to the repo, you can *pull* down updates
        - unlike Dropbox, git doesn't auto-sync the files in the repo

Connecting to ugrad

- On lab computer or your Windows laptop
  - Open PuTTy and connect to ugrad:
    - Open connection with hostname ugradx.cs.jhu.edu
- Mac: open Terminal application, then type the command
  ssh <your-username>@ugradx.cs.jhu.edu

- Alternatives to ugradx are ugrad1, ugrad2, . . . ugrad24

Our public file repository for this course

`https://bitbucket.org/cs220sum20/cs220sum20-public/`

- contains files shared with you for use in this course
- open a web browser and view this repo

Exercises

- On Piazza, find Resources section, then click Resources tab
- Now, do ex1-1: find the link under "Exercises" and follow the instructions
- questions? post on Piazza or attend office hours!