University of Southern California

Viterbi School of Engineering

EE599
Software Design and Optimization

Technologies
(Git, Linux)

Reference: Notes and Slides of Professor Bhaskar Krishnamachari and Mark Redekopp, Online Resources (White Papers, etc.)

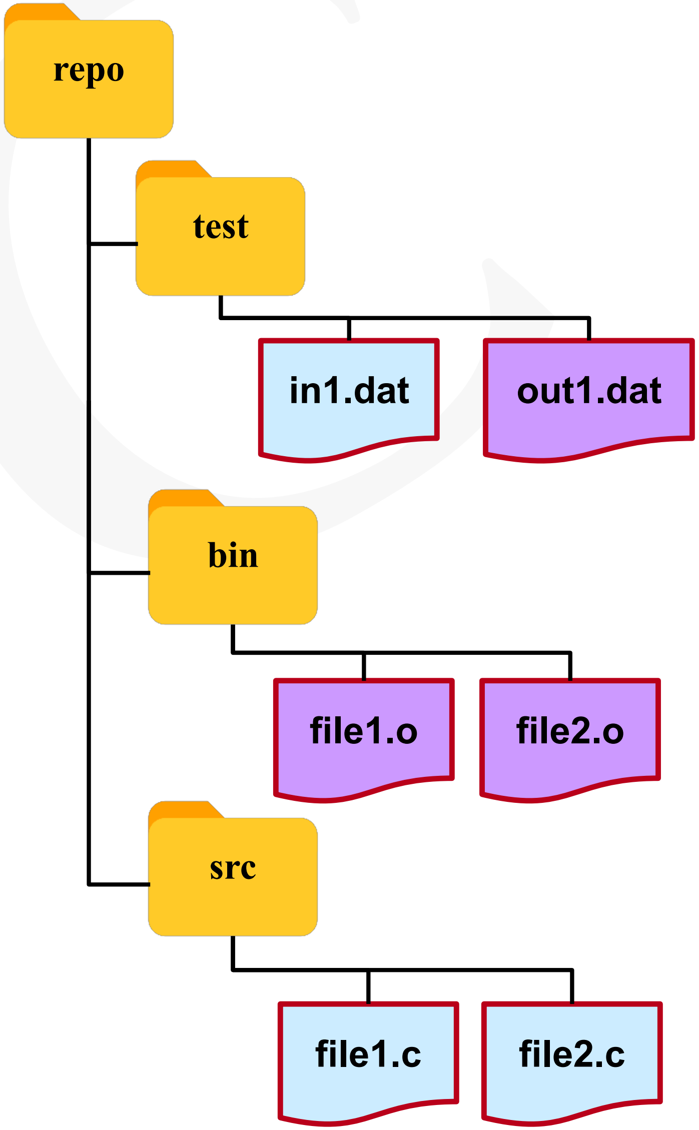
GIT AND GITHUB

Source/Version Control

- Have you ever made backups of backups of source files to save your code at various states of development (so you can recover to an earlier working version)?
- Have you ever worked on the same code with a partner and tried to integrate changes they made?
- These tasks can be painful without help
- Source/version control tools make this task easy
  - Allows one codebase (no separate folders or copies of files) that can be "checkpointed" (committed) at various times and then return back to a previous checkpoint/commit if desired
  - Can help merge differences between two versions of the same code
- Common source/version control tools are:
  - Git, Subversion, and a few older ones (cvs, rcs, clearcase, etc.)

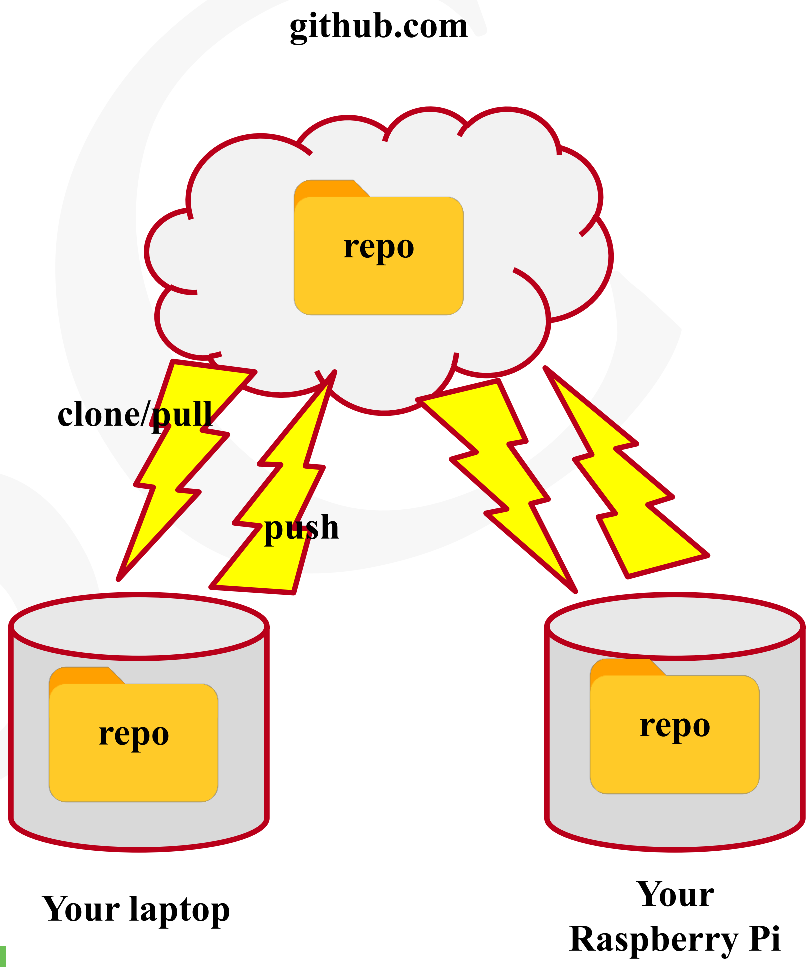
Repositories

- We generally organize our code and related files for a project in some folder
  - We will use the term "repository" for this *top-level* folder when it is under "version-control"
- Your repository can have some files that ARE version controlled...
  - Source code, Makefiles, input files
- ...and some that ARE NOT
  - Object files, executables, output files



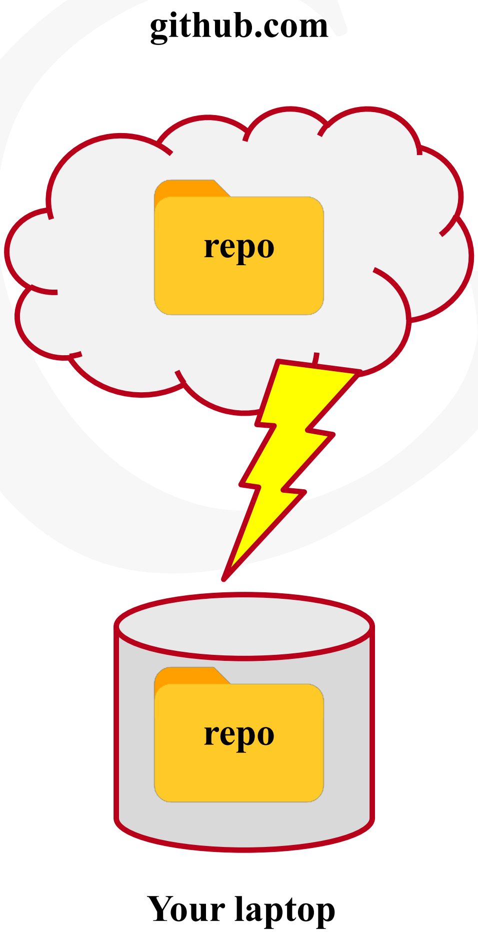
Git

- Git is a version control system
  - Stores "snapshots" of files (usually code) in a repository (think folder) at explicit points in time that you choose
    - No more making backup copies
  - Allows easy updates to a view of the code at some historical point in time
- Git is "distributed" (often via Github)
  - Allows the repository to exist on various machines and each store new updates (aka "commits")
  - Github holds the central repository
  - Updates can be communicated to each "clone" of the repository by "push"-ing updates to and "pull" updates from the central repository on Github



Cloning Repos

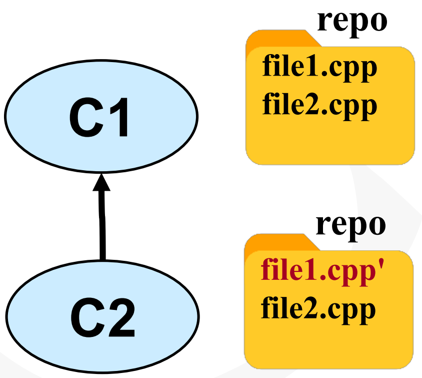
- Cloning a repo brings a copy of the specified repository onto your local machine
  - `git clone` *`url-of-repository`*
- You can now perform additions, modifications, and removals locally (without being connected)
- Allows the two repositories to be synchronized in both directions via `git push` and `git pull`



```
git clone git@github.com:usc-ee250-fall2018/Grove-Pi
```

Adds and Commits

- Repositories are updated by performing commits
- We first indicate all the files we want to commit by performing one or more adds via `git add`
  - Like adding things to your cart
- Then we perform a `git commit` of the added files
  - Like checking out
- Note: Don't add folders, just files...folder structure will be added automatically



Sample Sequence:

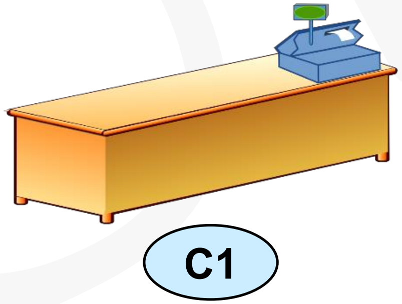
1

```
git add file*.cpp
```

file1.cpp
file2.cpp

2

```
git commit -m "Initial"
```

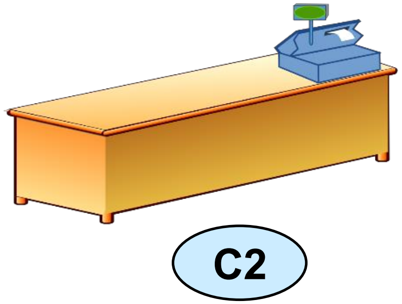


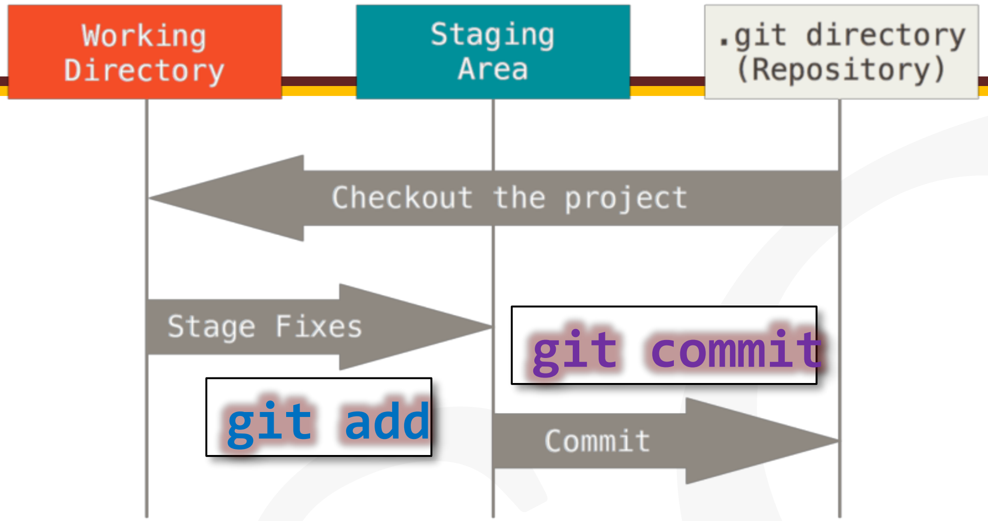
3

```
git add file1.cpp
```

4

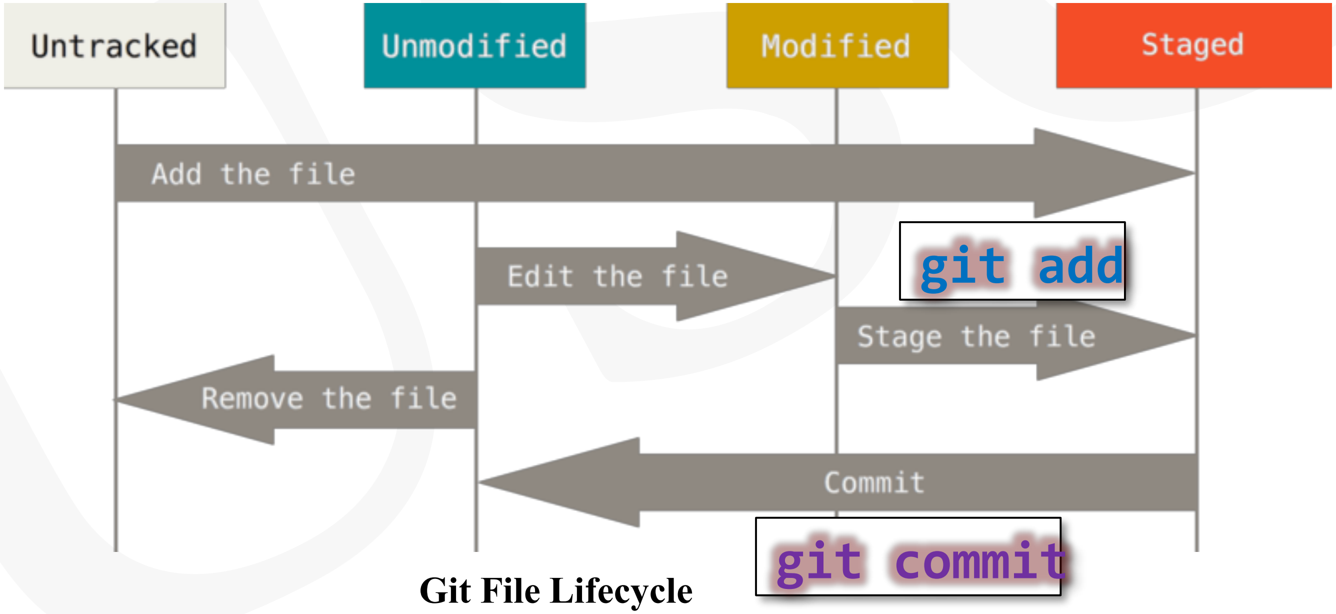
```
git commit -m "Updated"
```





Git "Locations"

https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

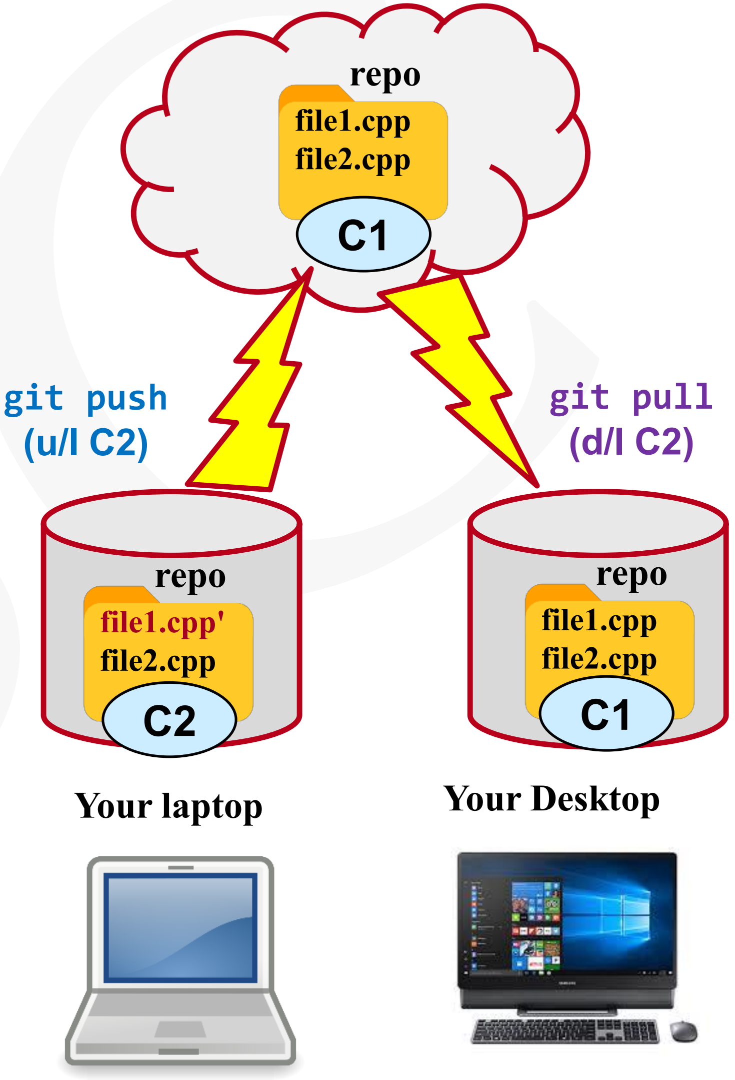


Git File Lifecycle

https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

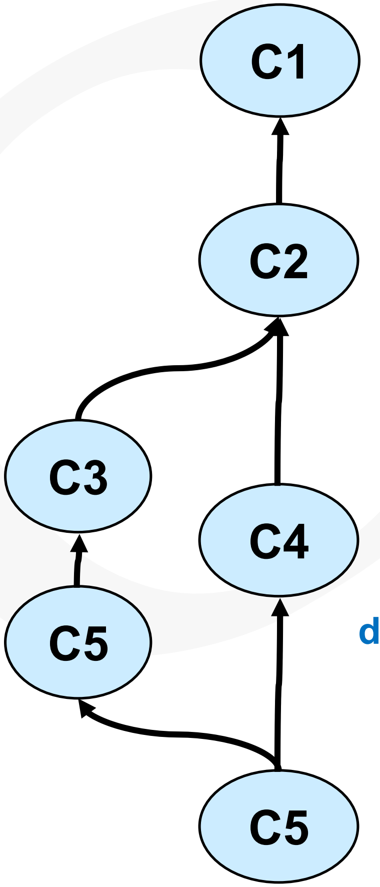
Push and Pull

- Suppose we make changes to our local repository
  - `git add file1.cpp`
  - `git commit -m "Added func2"`
- We upload the updates to the remote repository via a push operation
  - `git push`
- Another clone of the repository can download any updates from the remote repository via a pull operation
  - `git pull`



Branches Motivation

- Branches are useful when you are adding some new feature/fix, especially when other developers may also be doing the same by giving a separate sandbox to work in
- Branches allow you to
  - Grab the code from a particular starting point (i.e., commit)
  - Modify code, add, delete and commit
  - Merge the code back into the master branch



`feature1` branch (your updates for feature 1)

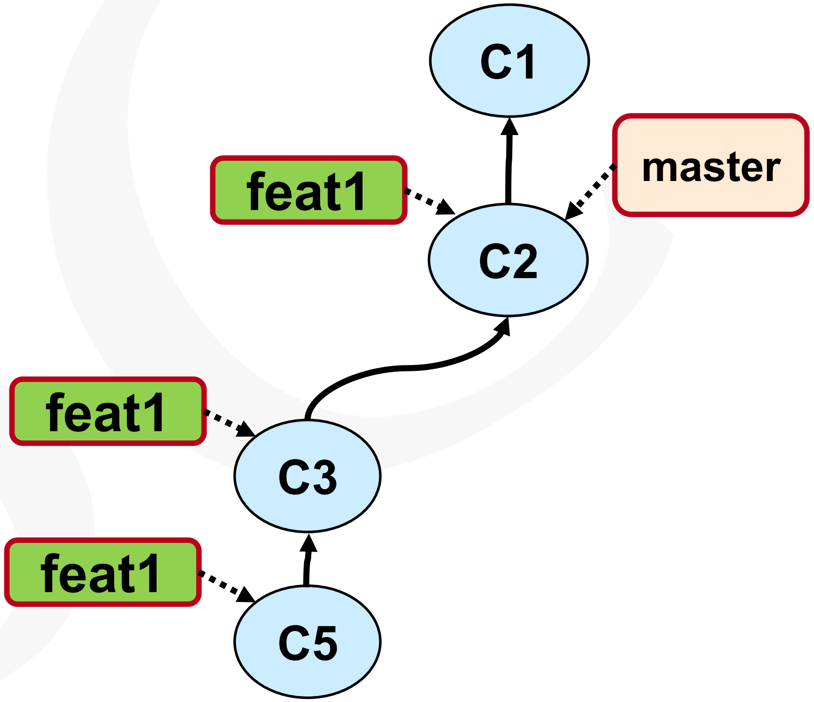
`master` branch (other developers or fixes to C2)

Branches (1)

- Each commit has one parent
- Branches are just names that can be associated with a commits
  - 'master' is the default branch
  - Created using:

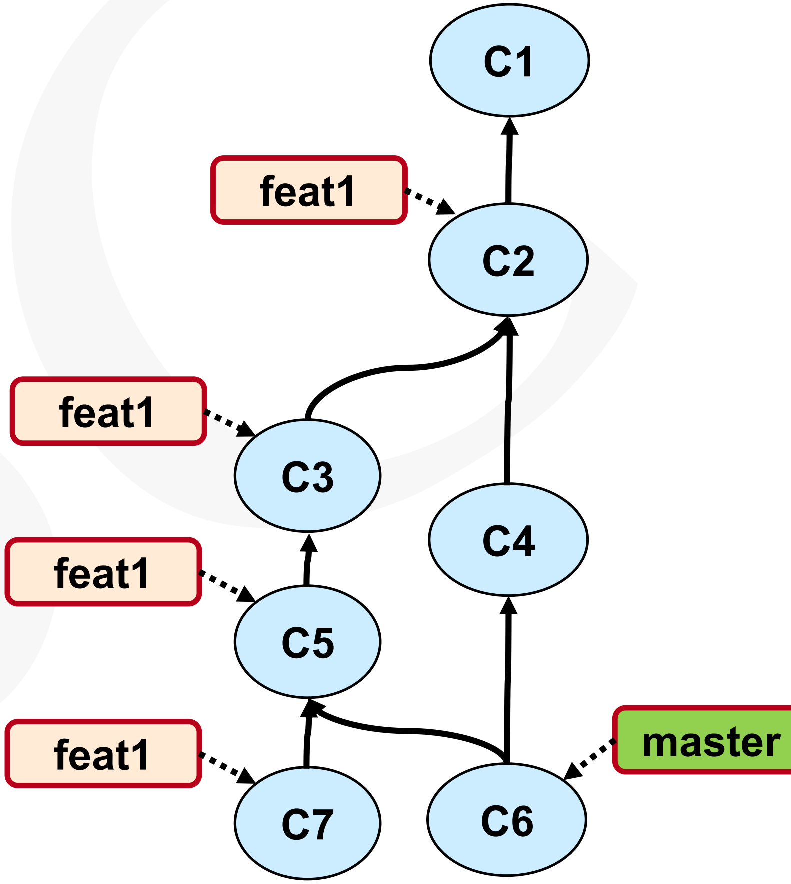
```
git checkout -b branch-name
```

- You can only be working on one particular branch at a time
- Any commits are applied to the current branch
- Example:
  - `git checkout -b feat1`
  - `git commit -m "Added part1"`
  - `git commit -m "Added part2"`



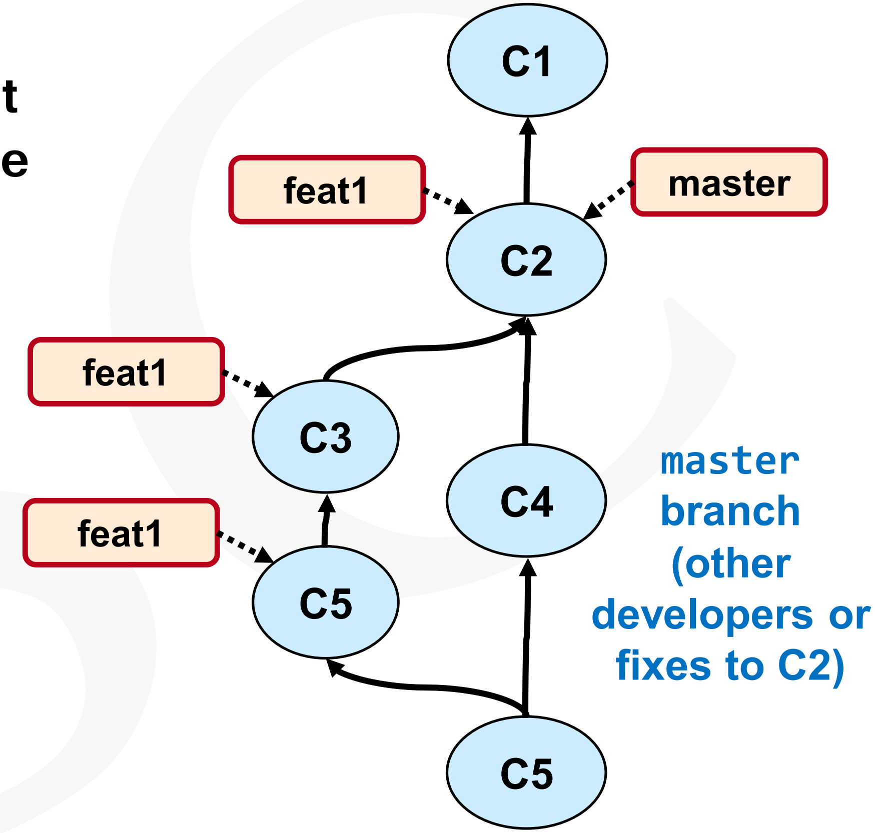
Branches and Merging

- We can switch between branches using `git checkout` ***branch-name***
- Example:
  - `git checkout master`
  - `git commit -m "Fix bug 1"`
- Two branches can then be merged together via:
  `git merge` ***branch-to-merge-in***
- A merge is a special commit with two "parents" and combines the code
- Example:
  - `git merge feat1`
- Note: You must be in the branch that will be updated with the code from the specified branch
  - The specified branch remains independent (you'd have to do another merge to sync both branches)
  - `git checkout feat1`
  - `git commit -m "Separate change"`



Conflicts

- If the merge encounters updates that it is not sure how to combine, it will leave the file in a conflicted state
- Can find conflicted files via:
  - `git status`
- Contents of conflicted files must be manually combined
  - Conflicted areas are highlight with `<<<<`, `====`, `>>>>` with the contents of each branch
  - Edit the file to your desired final contents
  - Then add and commit



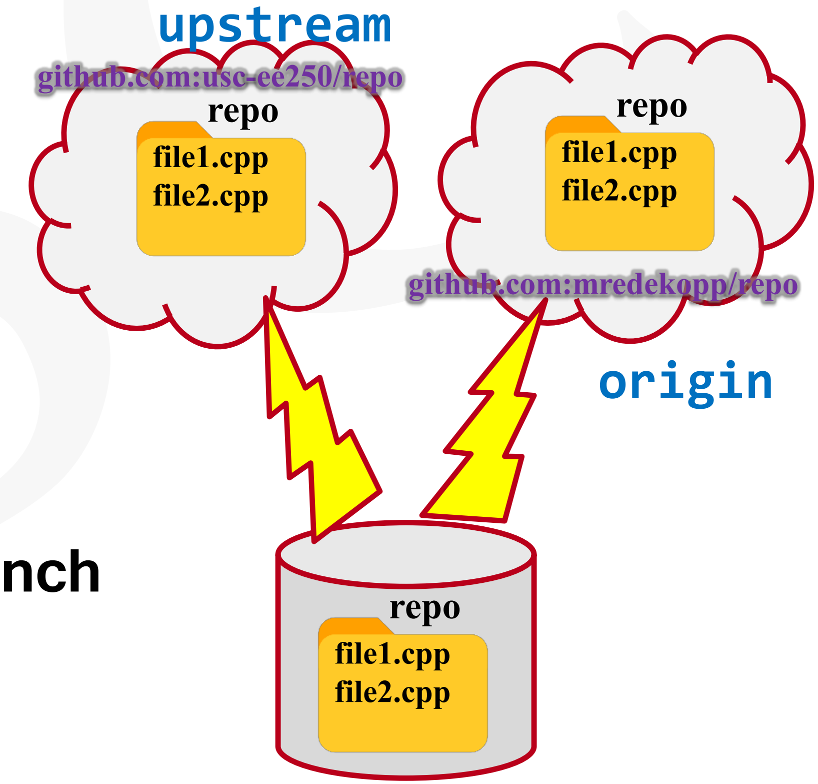
master branch (other developers or fixes to C2)

Sample Conflicted File

```
If you have questions, please
<<<<<<< HEAD
open an issue
=======
ask your question in IRC.
>>>>>>> feat1
```

Remotes

- Remotes are just like their name indicates: remote locations where we can push and pull (or fetch) data from
- To list remotes
  - `git remote -v`
- To add a remote
  - `git remote add` *name* *remote-url*
  - `origin` is the common name for the remote repo from which you cloned
  - A remote is just an association of a name to a repo URL
- To choose & push a particular branch to a remote
  - `git push -u` *remote* *local-branch*



Forks

- A fork is a "copy" of a repository
  - Essentially a new repo whose starting point is the current state of the original, "forked" repo
  - Allows changes to be made (like a branch) or starting a new project based on some current codebase
    - – If the original fork changes, there are means to pull those updates into your fork
  - It is possible to fork a fork ☺
- Example
  - The sensors we use have Python library support available on Github
  - We have forked that repo and made some changes for EE 250
  - You will then fork our repo (i.e. a fork of a fork) and modify it with your lab group
    - – If we make changes in our repo, you can easily bring them into your fork

Icons made by Dave Gandy from Flaticon is licensed by Creative Commons CC 3.0 BY

Upstreams

- Common definitions
  - upstream: The parent repository from which you forked
  - downstream: The forked ("child") repository (i.e., your repo)
- Common usage
  - The upstream fork can be thought of as just another remote
  - While the remote named origin usually refers to your fork on github, the remote named upstream usually refers to the parent of your fork
- Setting up access to the upstream fork
  - See https://help.github.com/articles/fork-a-repo/
  - `git remote -v`
  - `git remote add upstream` *`parent-fork-url`*
- Updating your code from the parent fork
  - `git fetch upstream`
  - `git checkout master` (can be skipped if you aren't using branches)
  - `git merge upstream/master`

Icons made by Dave Gandy from Flaticon is licensed by Creative Commons CC 3.0 BY

An Example

- Suppose we create a repo for you: `p1-ttrojan`
  - It comes preloaded (because of actions we took) with some code that was from our own repo: `p1-skel`
  - `git clone git@github.com:usc-ee250-Spring19/p1-ttrojan`
  - `cd p1-ttrojan`
  - `# You make changes; add, commit, push`
- Now we make changes to `p1-skel`, how can you get and merge those changes in?
  - `git remote -v       # list the remotes`
  - `git remote add upstream git@github.com:usc-ee250-fall2018/p1-skel`
  - `git fetch upstream  # d/l changes to a temp area`
  - `git checkout master # make sure you're in your master branch`
  - `git merge upstream/master  # Update your code`

Summary

- git add *file(s)*
  - Stage a file to be committed
- git commit -m "Change summary"
  - Makes a snapshot of the code you added
- git checkout -b *branch-name*
  - Create a branch and switch to it
- git pull
  - Download commits from your remote repository
- git push
  - Upload your local commits to the remote repository
- git checkout *branch-name*
  - Switch to a new branch
- git merge *other-branch-name*
  - Merge the commits from other-branch-name into current branch
- HEAD is synonymous with the current branch's latest commit
- origin is usually the remote name for your repo on github
- upstream is usually the remote your repo was forked from (must be added)

Helpful Links

- https://help.github.com/
- http://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf
- https://learngitbranching.js.org/
  - Main: Level 1 - Intro to Git Commits
  - Main: Level 2 - Ramping Up
  - Remotes: Level 1: Push & Pull – Git Remotes
  - Remotes: Level 2: To Origin and Beyond
- https://services.github.com/on-demand/downloads/github-git-cheat-sheet/ (web version)
- https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf (print version)

LINUX

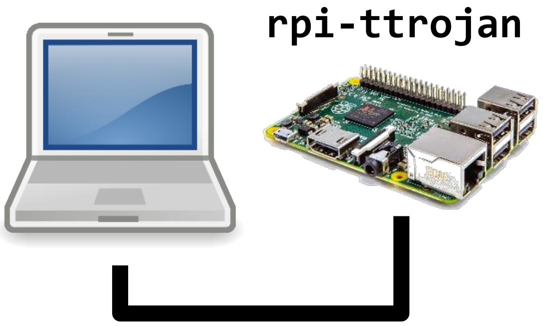
The Linux OS

- Based on the Unix operating system
- Developed as an open-source ("free") alternative by Linux Torvalds and several others starting in 1991
- Commonly used in industry and in embedded devices

ssh Connections

- ssh stands for Secure Shell
  - Encrypted protocol to run a terminal (command line) on a remote server
- Usage
  - `$ ssh `*`username`*`@server`
  - `$ ssh ttrojan@aludra.usc.edu`
    - Use your USC password first, then try 10-digit ID
  - `$ ssh pi@rpi-ttrojan`

This Photo by Unknown Author is licensed under CC BY-NC-SA



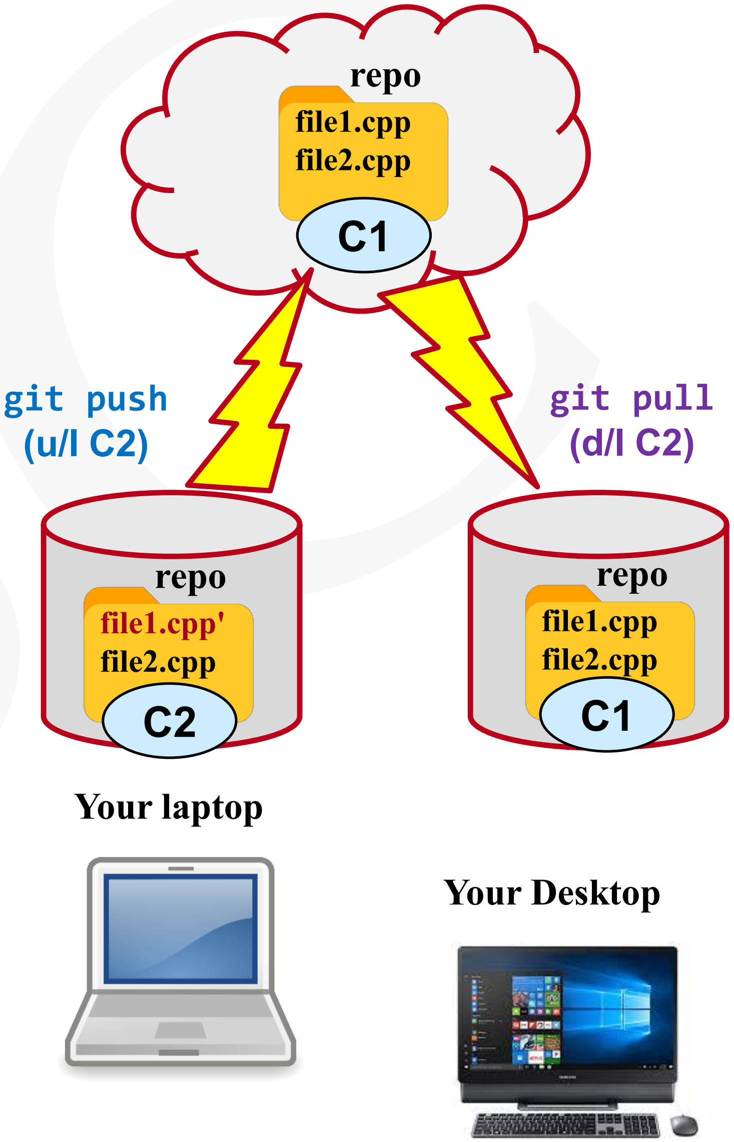
Terminal/Command Line

- No more GUI!
  - ssh connections provide only command line interfaces to the remote machine
- Have to be comfortable navigating the file system, using command line utilities and text-based editors
  - Paths (e.g. ../ee.../Grove-Pi/examples)
  - Utilities (cd, mv, cp, rm, mkdir, cat, more, grep)
  - Text-based editors (vi, emacs, nano)

How Will We Transfer Files

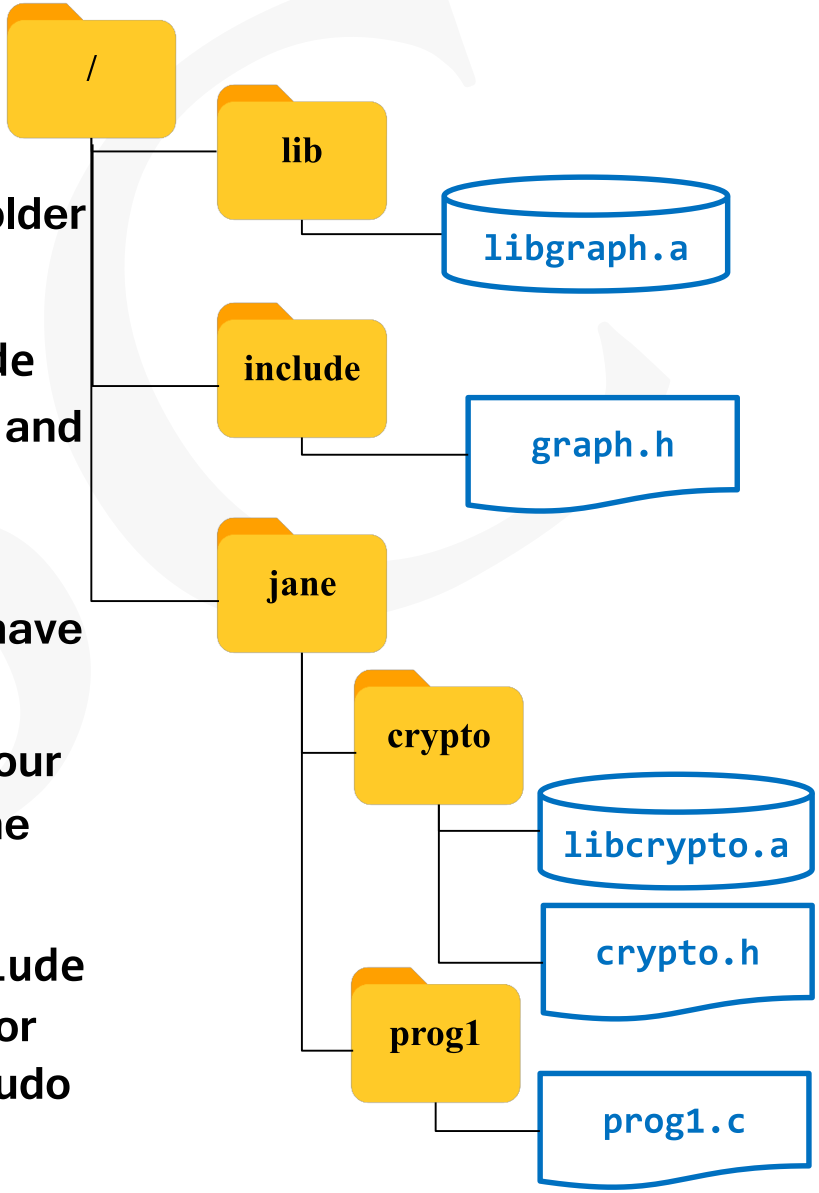
- With git
  - add/commit/push on one machine
  - pull on the other
- Good to know but should not be used in place of git:

scp
  - Same as 'cp' (secure cp) but copies to/from current machine to another network machine over a secure connection
  - scp current_file dest_loc
    - Current_file or dest_location can be on a remote machine which requires user@machine syntax before file location
  - Examples:
  - scp username@aludra.usc.edu:temp/hi.txt .
  - scp *.cpp username@aludra.usc.edu:temp/

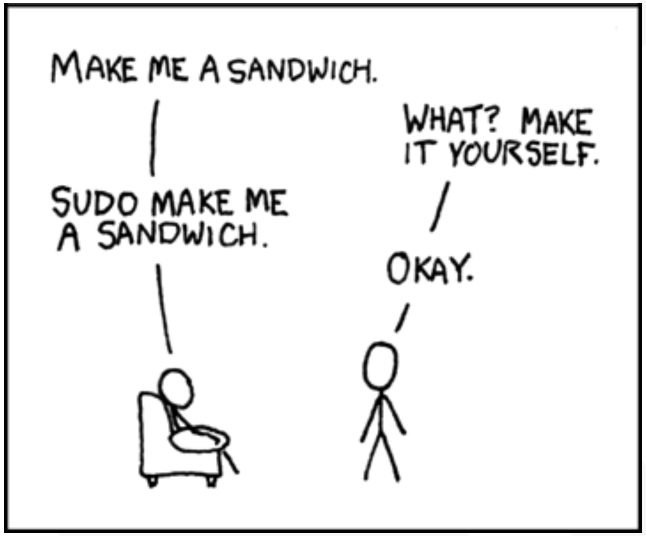


Access Control and sudo

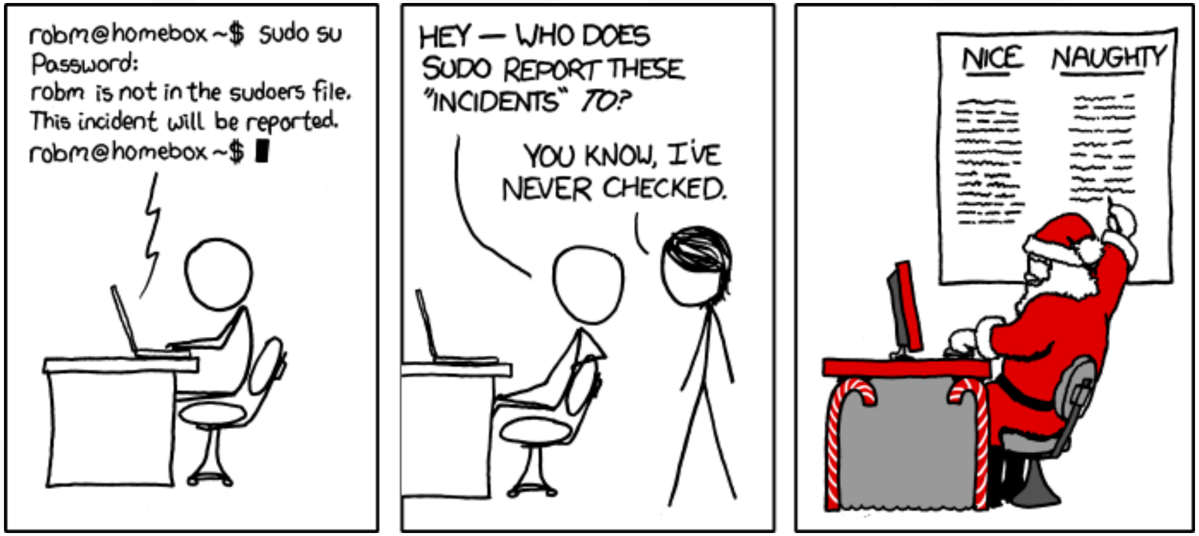
- Linux enforces access control per user
  - You have access to /home/*username*
  - You likely cannot copy from your home folder to /include
    - `$ cp ~/crypto/crypto.h /include`
  - The system is represented by user `root` and has superuser/supervisor access
- The magic of `sudo`
  - Admins can configure a user account to have supervisor privileges
  - If you have supervisor access, precede your command with sudo and it will execute the command with elevated privileges
  - `$ sudo cp ~/crypto/crypto.h /include`
- Some labs will require you to run commands (for example to use a USB device) preceded with sudo



sudo Humor



https://imgs.xkcd.com/comics/sandwich.png



```
robm@homebox~$ sudo su
Password:
robm is not in the sudoers file.
This incident will be reported.
robm@homebox~$ ▌
```

HEY — WHO DOES SUDO REPORT THESE "INCIDENTS" TO?
YOU KNOW, I'VE NEVER CHECKED.
NICE
NAUGHTY

https://imgs.xkcd.com/comics/incident.png

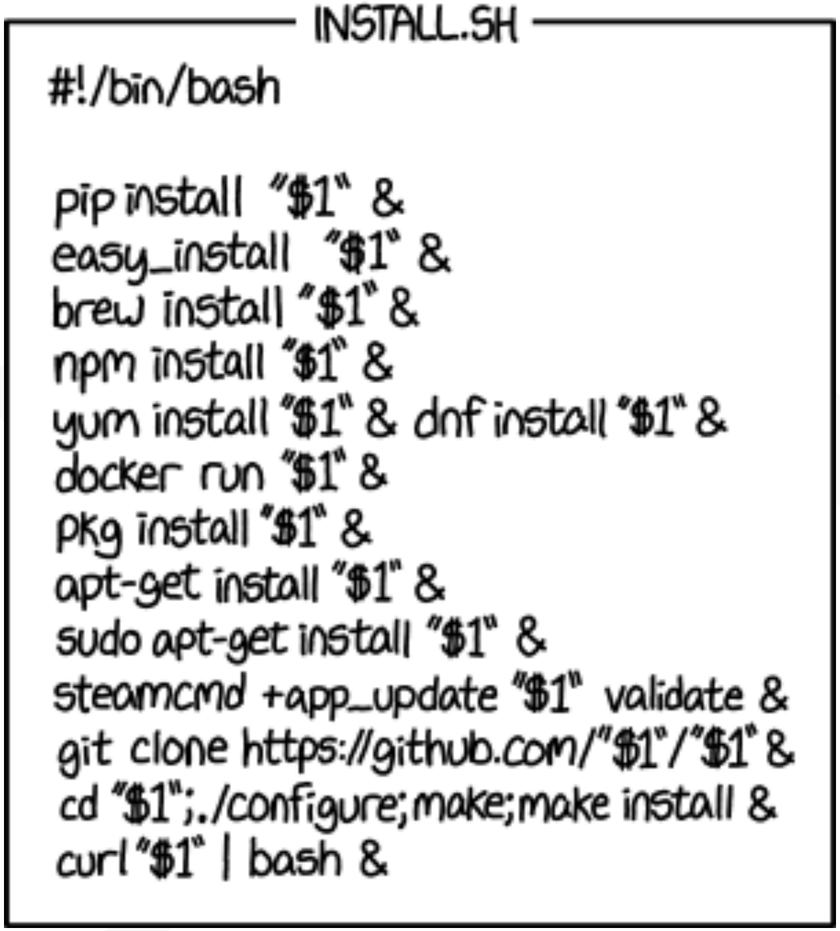
Installing Linux Packages/Programs

- Ubuntu maintains servers that have distributions of programs/packages available for installation
- Many packages depend on other packages and so installing one requires installing many others
- Ubuntu makes this easier with installation utilities and package manager (`apt-get`)
- Common syntax:
  - sudo apt-get install *package-name*
  - Examples:
    - `sudo apt-get install build-essential`
      - Installs g++ compiler and other
    - `sudo apt-get install git`
    - `sudo apt-get install vim`

Installing Python Packages

- **Just like apt-get helps with distribution and installation of programs for Ubuntu, pip3 downloads, installs, and configures Python packages/libraries for python3**
  - `$ pip3 install` *`package-name`*

More "Humor"



INSTALL.SH

```
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" & dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1";./configure;make;make install &
curl "$1" | bash &
```

https://imgs.xkcd.com/comics/universal_install_script_2x.png

Environment Variables

- Contain values that can be accessed by other programs that provide system and other info
  - `PATH`
    - Where to look for executables
  - `LD_LIBRARY_PATH`
    - Where to look for libraries
- Set a variable
  - In a terminal/shell with `export` command
    - `export VARIABLE=VALUE`
  - In a Makefile
    - `VARIABLE=VALUE`
- Use with `$VARIABLE` in shell
- Use with `$(VARIABLE)` in Makefile

Makefile

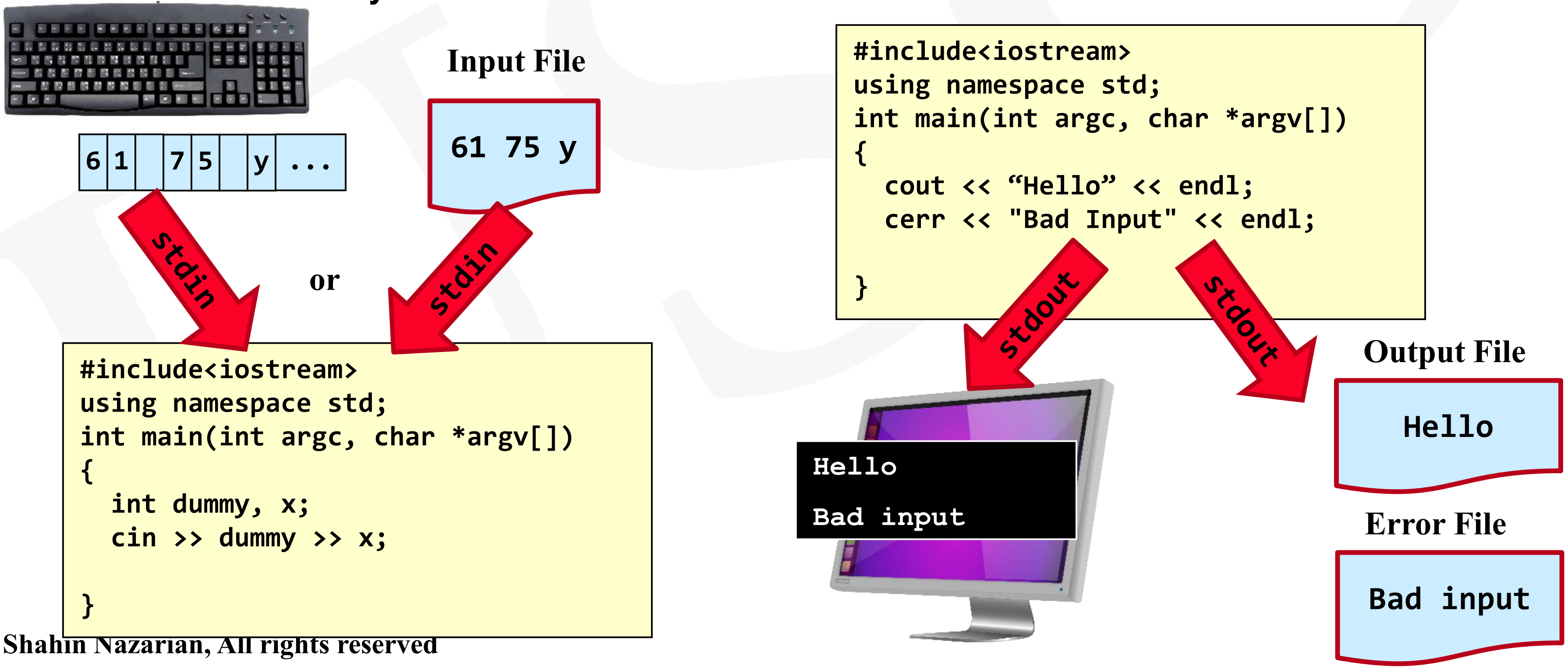
```
OBJECTS = lcd.o adc.o proj.o
CC = gcc

proj.elf: $(OBJECTS)
    $(CC) -o proj.elf $(OBJECTS)

lcd.o: lcd.h lcd.c
    $(CC) -c -Wall lcd.c -o lcd.o
```

Stdin, stdout, stderr

- Programs pull input from a stream known as 'stdin' (standard input)
  - It is usually the stream coming from the keyboard but can be "redirected" to pull data from a file
- Programs output stream known as 'stdout' (standard output)
  - It is usually directed to the screen/terminal but can be "redirected" to a file
- Programs have a 2nd output stream known as 'stderr' (standard error)
  - It too is usually directed to the screen/terminal but can be "redirected" to a file



```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  int dummy, x;
  cin >> dummy >> x;

}
```

```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  cout << “Hello” << endl;
  cerr << "Bad Input" << endl;

}
```

```
Hello
Bad input
```

Redirection & Pipes

- '<' redirect contents of a file as input to program
  - ./prog1 arg1 arg2 < input.txt
- '>' redirect program output to a file
  - ./prog1 arg1 > output.txt
- '>&' redirect stderr to a file
  - g++ -o test test.cpp >& compile.log
- '|' pipe output of first program to second
  - grep "day" re.txt l wc -l
- 'l&' pipe stderr of first program to second
  - g++ -o test test.cpp l& grep error

Summary

- You should...
  - Feel generally comfortable writing basic Python scripts that manipulate strings, integers, etc. and call library functions
  - Understand the basic git add, commit, push, pull process
  - Understand branches/remotes
  - Know to use sudo apt-get install or pip3 to install Linux packages or python-specific packages