CS 241 Honors
Class Introduction

CS 296-41 Course Staff

University of Illinois Urbana-Champaign

January 24, 2017

Course Structure

Why are we here?

To provide an opportunity to learn about and explore systems programming at a deeper level than in CS 241

What is systems programming?

- Process control
- Inter-process communication
- Parallelism and synchronization
- Memory management
- Filesystems
- Networking
- Security

Course Structure

The course largely consists of a lecture series and a semester-long project:

Lectures

- Tuesday, 6-6:50pm, 0216 Siebel
- There is an attendance grade, but...
- We drop 3 absences without question (interview, medical, etc.)

Projects

- Teams consist of 1-3 people (larger teams allowed on individual basis)
- Each team is assigned a mentor from the course staff
- Must be technically rigorous
- Must relate to systems programming
- *No games allowed* (unless you convince us otherwise)

Course Structure

Grade Breakdown

- 25% – Lecture attendance
- 75% – Semester project
  - 5% – Project Proposal
  - 25% – Weekly check-ins with mentor
  - 35% – Final deliverables (code, project website, etc.)
  - 10% – Final presentation

The minimum grade thresholds for this class are the same as CS 241

Course Staff

Name	NetID	Interests
Robert Andrews	rgandre2	Algorithms, Complexity Theory, Parallel Computing
Aneesh Durg	durg2	Computer Vision, AI
Kevin Hong	khong18	Concurrency, IPC, Statistics, Machine Learning
Ben Kurtovic	kurtovc2	Operating Systems, Networking, Security
Bhuvan Venkatesh	bvenkat2	Cloud Computing
Jonathan Wexler	jwexler2	Parallelism, Networking
Brian Zhou	bwzhou2	Full-stack development, Networking

If there is someone you are interested in working with, feel free to reach out via email!

Student Demographics

We have 38 Computer Science majors in this class

- 24 in Computer Science
- 8 in CS + Math
- 5 in CS + Stats
- 1 in CS + Linguistics

There are 2 of you not in Computer Science!

- 1 in New Media (FAA)
- 1 in DGS

Project Ideas

Past successful projects have included:

- A 3D scanner built from a Microsoft Kinect
- A top-down multiplayer space shooter with a custom physics engine
- A custom debugger
- An earliest-deadline-first scheduler
- A garbage collector for C
- A general-purpose network encryption utility
- A chess engine
- A theremin built in software

You can see a larger collection of successful past projects at `cs241.cs.illinois.edu/past_projects.html`.

Course Structure

Why are we here?

To provide an opportunity to learn about and explore systems programming at a deeper level than in CS 241

What is systems programming?

- Process control
- Inter-process communication
- Parallelism and synchronization
- Memory management
- Filesystems
- Networking
- Security

Course Structure

Lecture topics include:

- Advanced parallel programming
- Filesystems
- Security
- Distributed systems
- Memory management and garbage collection
- Debugging strategies
- Cloud computing
- Lock-free data structures
- TCP

This list is not set in stone; if you want to learn about something, tell us!

Software Development

Let's talk about good software development practices...

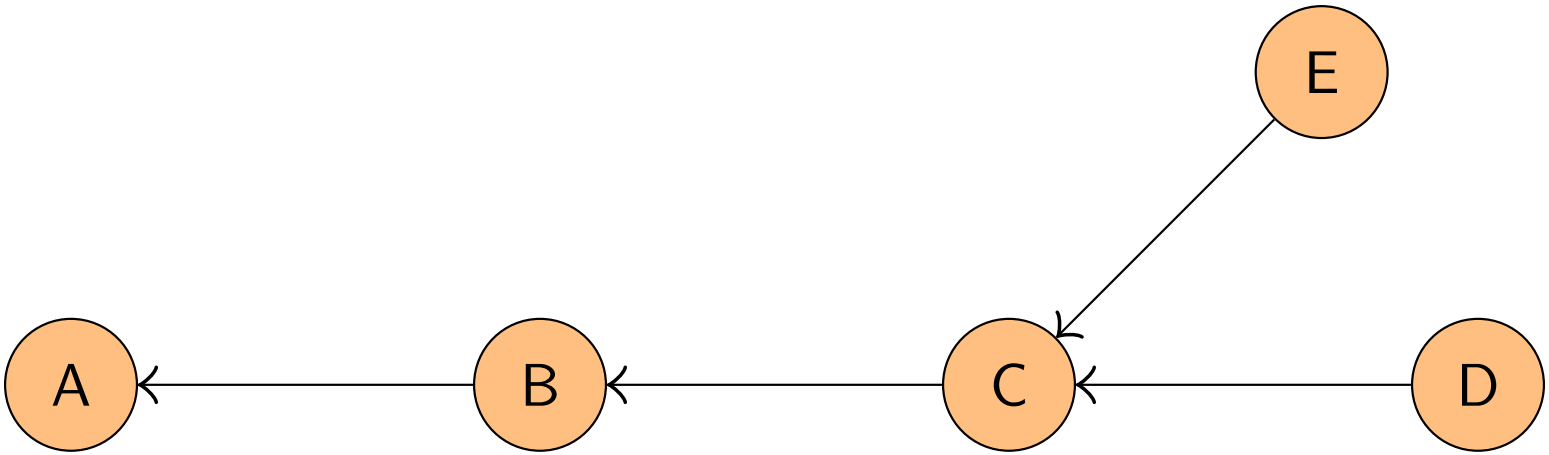
Version Control With Git

Why use version control?

Using version control allows you to easily...

- Collaborate with others
- Track multiple versions of your project
- Revert to a previous iteration of the project
- Develop new features without fear of breaking existing features

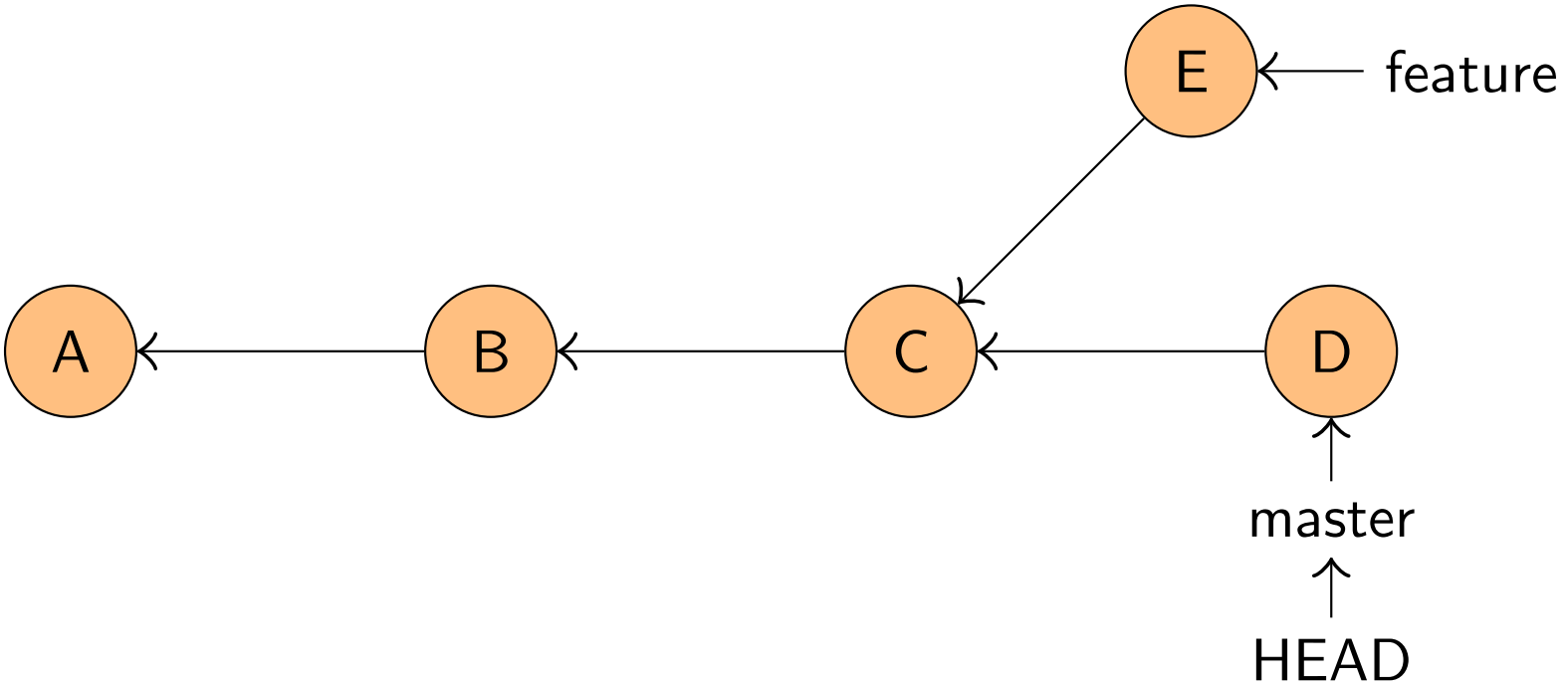
Version Control With Git



Git's internal model

- Git models a repository as a directed graph
- Each node in the graph corresponds to a revision of the project
- Edges are directed from child commit to parent commit

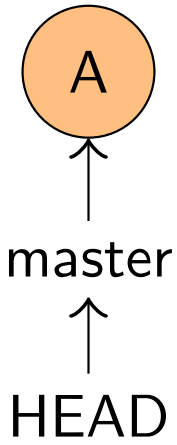
Version Control With Git



Git's internal model

- Git also maintains a collections of *heads* (AKA *branches*), which are references to commits
- By default, a repository will start work on the *master* branch
- *HEAD* points to the currently checkout out commit

Version Control With Git



Initializing a repository

- Run `git init` inside a directory to initialize a Git repository inside that directory
- After the first commit, both *master* and *HEAD* will point to that initial commit

Version Control With Git

Making a commit in Git involves two steps: staging and committing

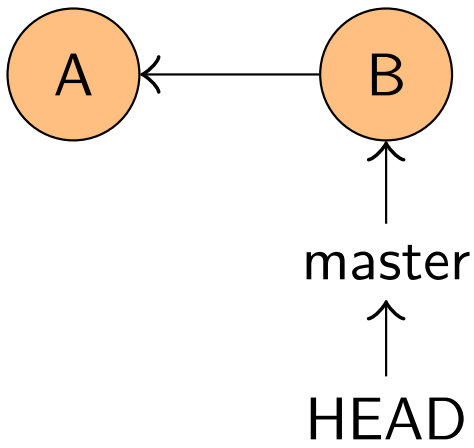
Staging

Using `git add`, add files to the staging area to be used in the next commit. Alternatively, you can use `git add -i` to stage things in interactive mode, which lets you work at the granularity of *hunks* (i.e., blocks of code in a file) rather than staging a whole file at once.

Committing

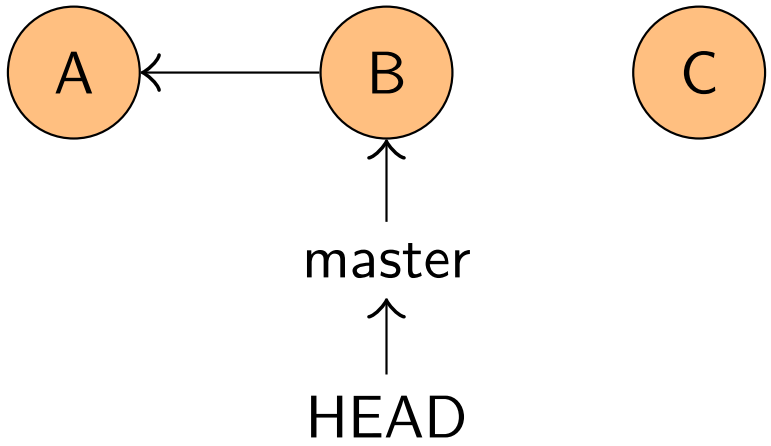
Once you've finished staging everything for commit, run `git commit`. You will be prompted to enter a commit message, after which your commit will be added to the repository. Strive to make your commit messages descriptive and split logical chunks of work into different commits!

Version Control With Git



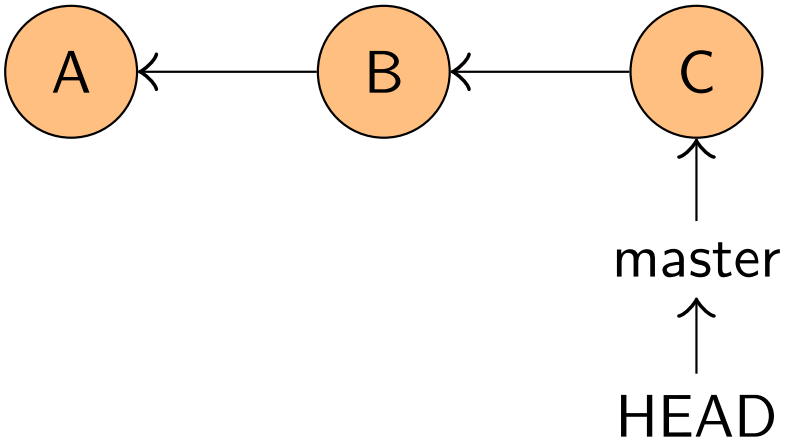
Suppose our project consists of two commits, A and B, where A is the initial commit and B is the most up-to-date revision of the repository.

Version Control With Git



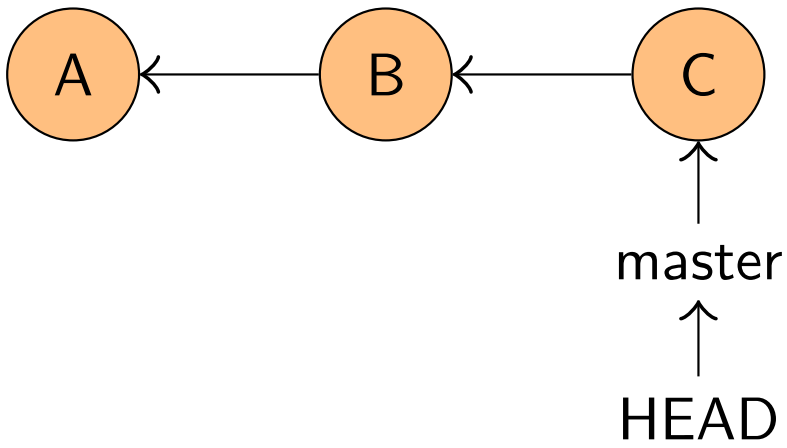
Suppose our project consists of two commits, A and B, where A is the initial commit and B is the most up-to-date revision of the repository. Now we stage a new commit, C.

Version Control With Git



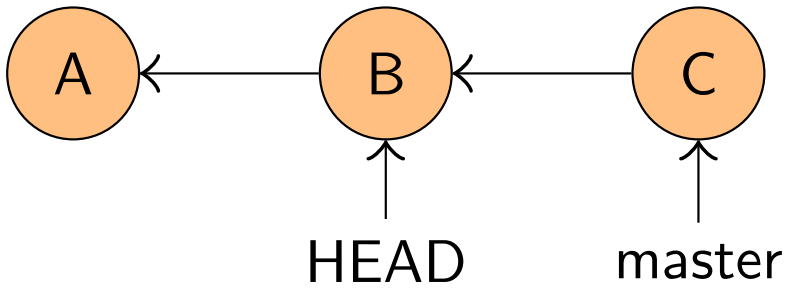
Suppose our project consists of two commits, *A* and *B*, where *A* is the initial commit and *B* is the most up-to-date revision of the repository.
Now we stage a new commit, *C*.
After running `git commit`, *master* and *HEAD* are updated to point to *C* and *C* has its parent set to *B*.

Version Control With Git



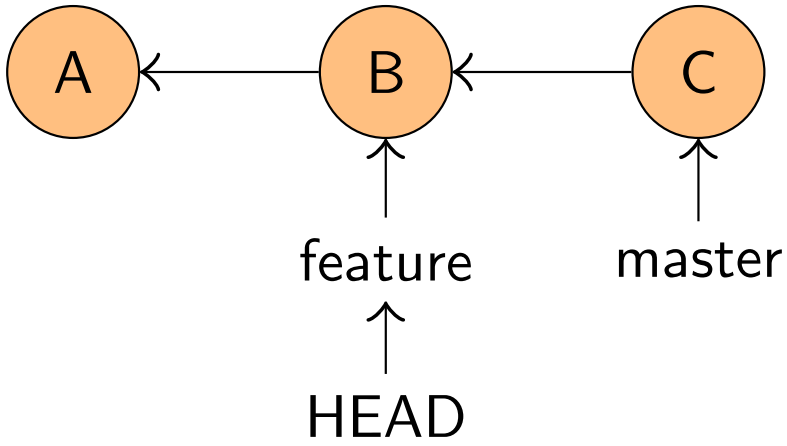
Suppose now that we want to develop a new feature, D, but starting from the state of the project at B.

Version Control With Git



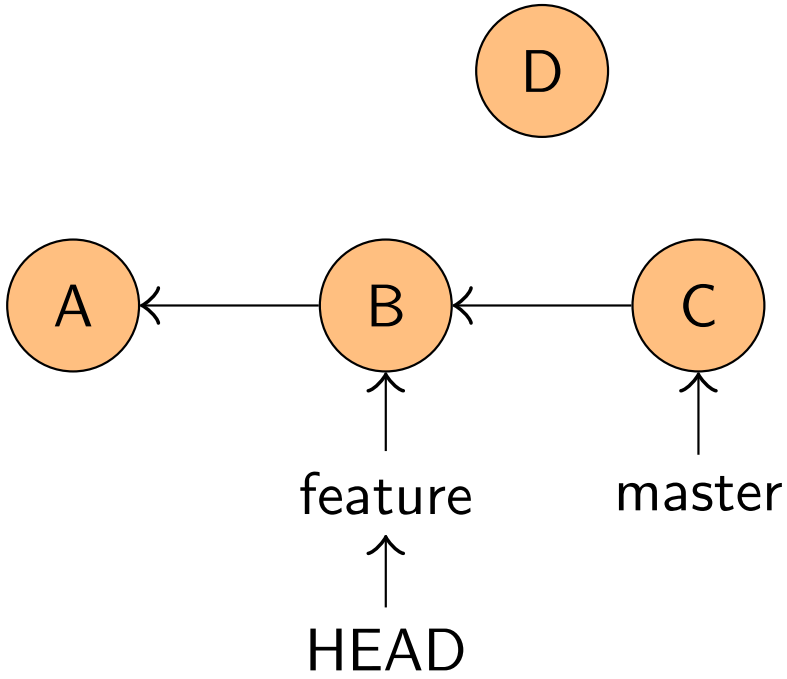
Suppose now that we want to develop a new feature, D, but starting from the state of the project at B.
First, we need to `git checkout` commit B.

Version Control With Git



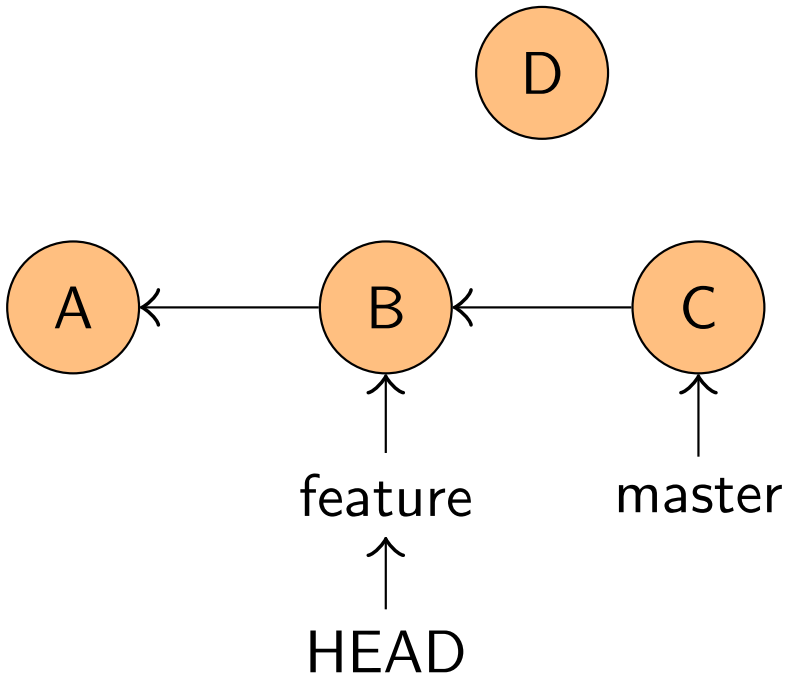
Suppose now that we want to develop a new feature, D, but starting from the state of the project at B.
First, we need to `git checkout` commit B.
Next, we'll start a new branch at B using `git branch feature`.

Version Control With Git



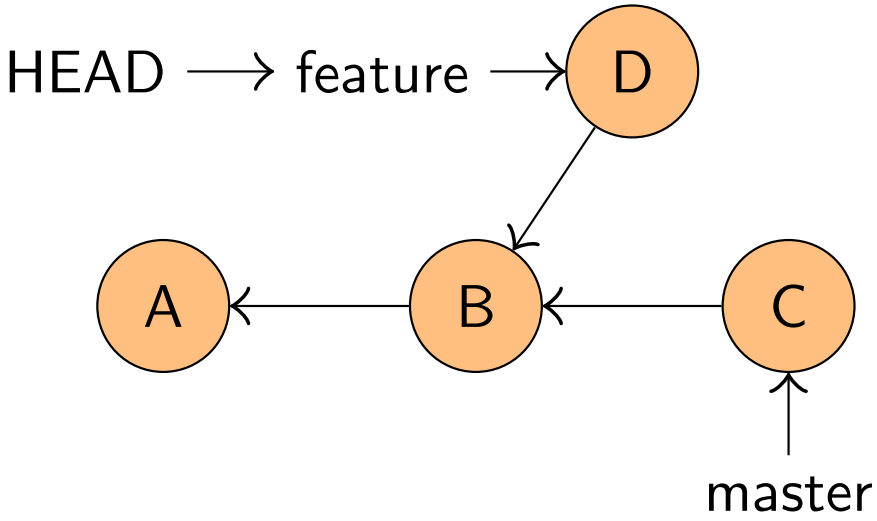
Suppose now that we want to develop a new feature, *D*, but starting from the state of the project at *B*.
First, we need to `git checkout` commit *B*.
Next, we'll start a new branch at *B* using `git branch feature`.
Then, we develop the features needed in *D*.

Version Control With Git



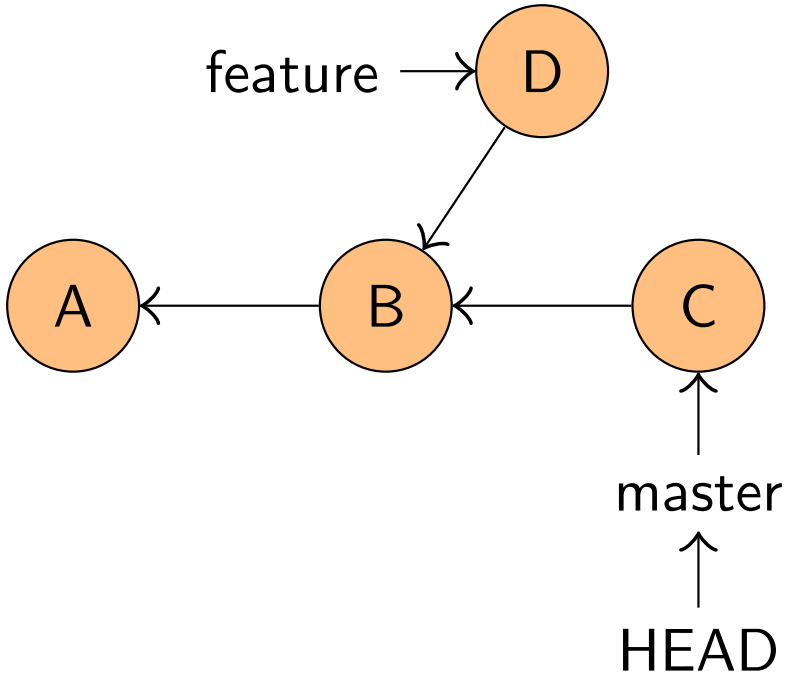
Once we have *D* finished and staged, we run `git commit`.

Version Control With Git



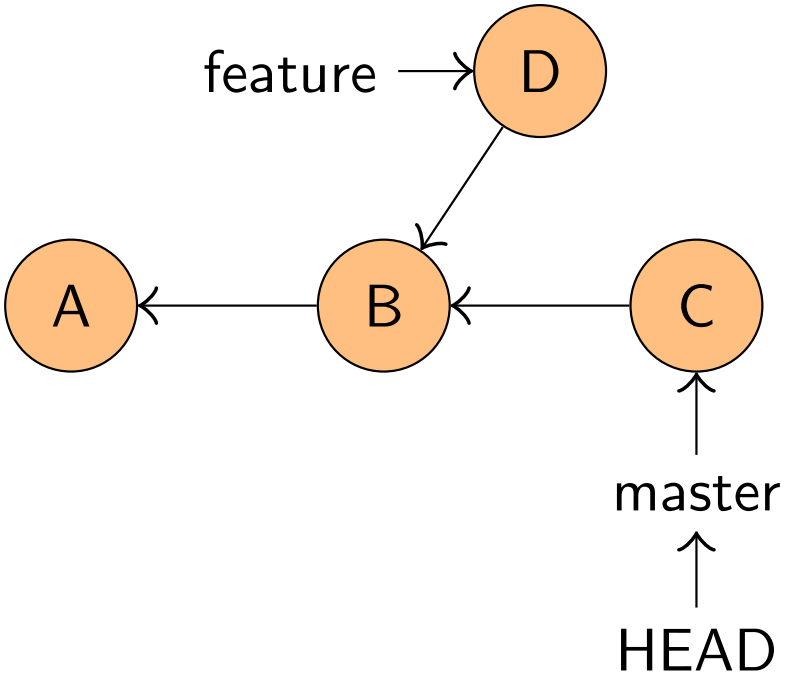
Once we have *D* finished and staged, we run `git commit`.
This creates the commit *D* and advances both *HEAD* and the branch *feature*.

Version Control With Git



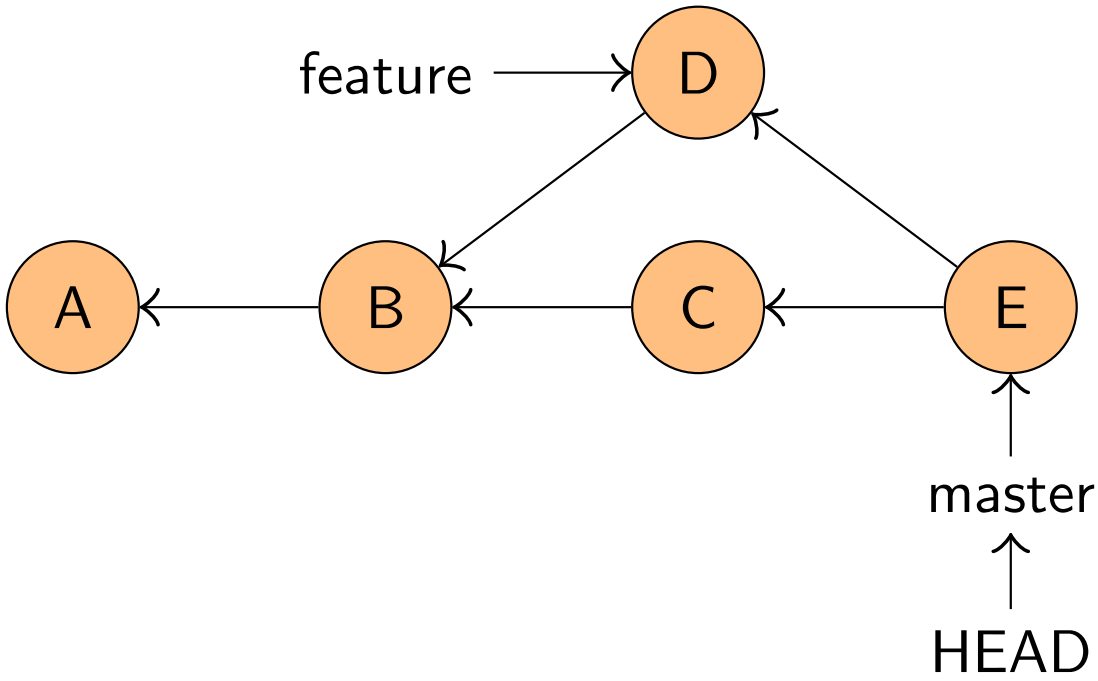
Once we have *D* finished and staged, we run `git commit`.
This creates the commit *D* and advances both *HEAD* and the branch *feature*.
If we want to checkout *master* again, just run `git checkout master`.

Version Control With Git



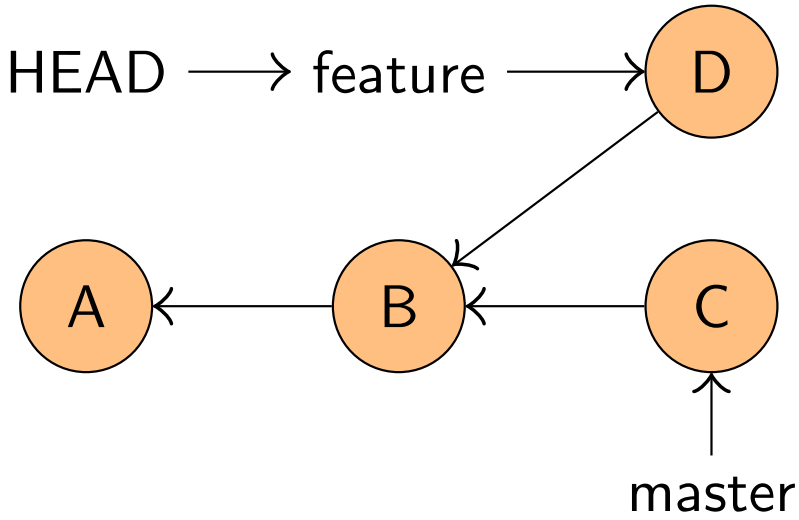
Let's say we want to combine the work done in *master* and *feature*. Git can do this automagically for us via `git merge`. If we have *master* checked out, running `git merge feature` merges *feature* into *master*.

Version Control With Git



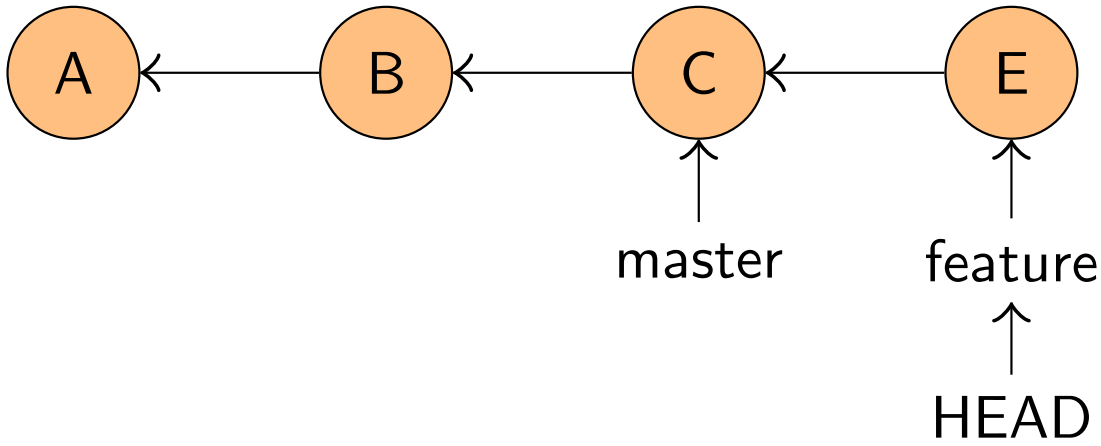
Doing this creates a merge commit E which contains the changes made in both C and D and advances the *master* branch.
If Git cannot automatically do the merge for you, you will have to resolve the merge conflicts manually by selecting what code you want to keep from both *feature* and *master*.

Version Control With Git



Alternatively, we can rebase *feature* onto *master* using `git rebase`. If we are working on *feature* and want to rebase onto *master*, we simply run `git rebase master`. This will remove *feature* from the least common ancestor of *feature* and *master* and create a copy of that subtree from *master*.

Version Control With Git



This avoids creating a merge commit, but it removes *D* from the project history and replaces it with a new commit *E*.
Notice that rebasing changes the history of the project! Be careful that your rebase doesn't remove a part of the history that other people are basing their work off of.

Version Control With Git

Finally, a few commands useful for working with remote repositories (e.g. a GitHub repository):

- `git fetch`: update your local repository with new commits made on the remote repository
- `git pull`: fetch remote commits and then merge the remote version of your current branch with the local version of your current branch
- `git push`: update the remote repository with your new local history

Writing Readable Code

- When you write solutions for an MP, the only other entity looking at your code is a machine
- Now, you are writing code that other *people* have to read!
- Some general guidelines:
  - Name your variables and functions so that it's obvious what they're used for
  - If someone else can't tell what your code is doing at a glance, write comments to explain it
  - Code in whatever style you want, but make sure your style is consistent
- Keep in mind that your teammates are not mind-readers; they cannot guess what you were thinking when you were writing your code

Team Communication

- If you are working on a team, you will likely need to meet as a team (outside of your mentoring meetings) to determine who is doing what
- Consider managing units of work on your project using the issue tracker on GitHub
  - Each logical unit of work should have its own ticket
  - Using an issue tracker allows you to keep track of any conversation pertaining to a particular feature
  - Assigning team members individual tickets can help organize collaboration
- If two members are developing features that interact with one another, make sure you are coordinating your work so that your features can work together properly

Time Management

- These projects are long-term and fundamentally different from most other assignments you've done
- It's very easy to fall behind, so make sure you're making progress every week
- Suggestion: make deadlines for yourself over the course of the semester! (or ask your mentor for deadline suggestions)
- It's much easier to make a few hours of progress every week instead of trying to make 40+ hours of progress in a single week