CptS 322- Programming Language Design

Version Control and git Basics

Instructor: Sakire Arslan Ay

Fall 2021

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Outline

- Review basic concepts
  - Commits, diffs, merges
- Good practices
  - Committing
  - Branch management
- Scenarios

Version Control

- VC tracks multiple versions of your project
  - Different releases
  - Different versions of the same release
  - In an extreme, every time you edit the project

Use Cases of Version Control

- Compare current version with older
  - To see what changed, e.g., since a bug was introduced
- Find out when/why/who edited a certain line
- Revert to an older version
  - To discard recent changes
  - To test when was a bug introduced

- Allow multiple teams to develop concurrently
- Allow multiple versions to be developed concurrently

Version Control – git Basics

W

Arrow represents a set of changes and points to parent

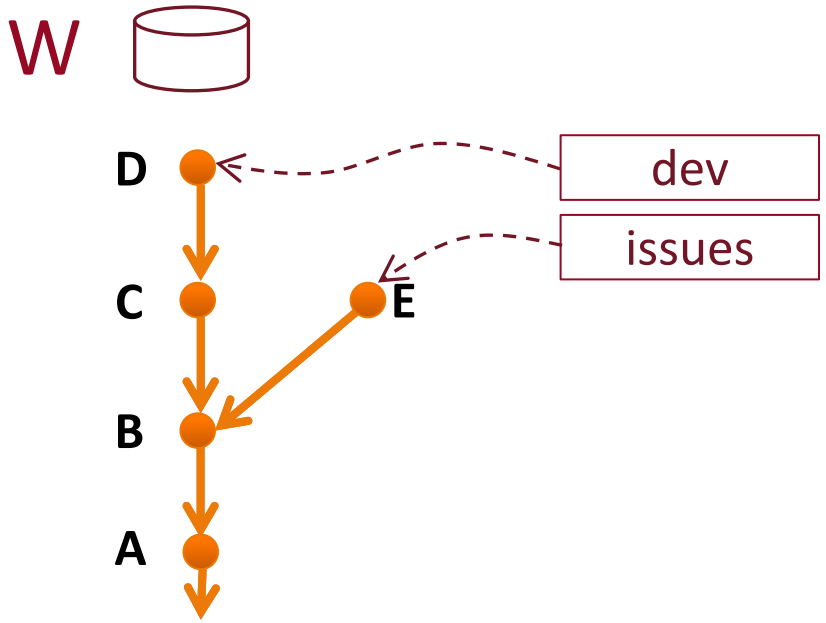
- Working directory (W)
  - Set of files/dirs under version control
  - This is where you make changes
- Commit
  - A snapshot of W, along with date/ author/message, and parent commit(s)
  - Parent: (logically) previous commit
- Repository
  - A database with commits
  - A directed-acyclic graph

git Revision Names

Arrow represents a set of changes and points to parent

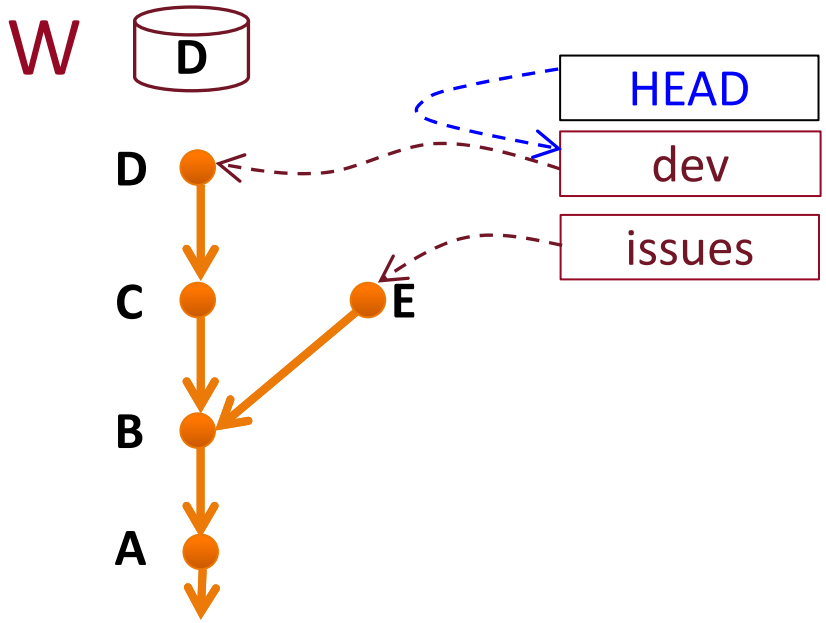
- git names revisions with a 40-char hexadecimal string
- Typically first few (5) chars are unique and enough for commands

git References



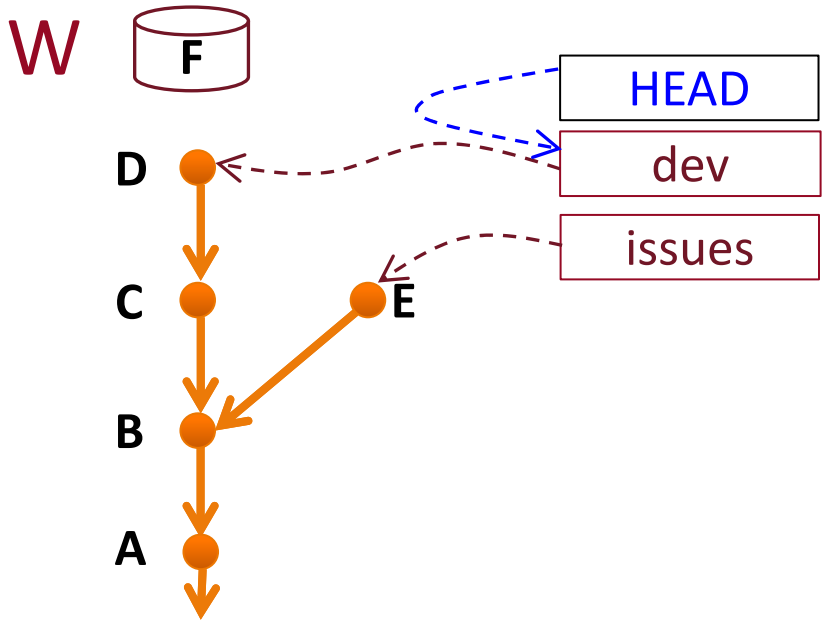
- git maintains a table of “references” (a.k.a. “refs”) which refer to commits in the repo
  - Branches are represented as “refs” (e.g., dev, fix)
  - There are also “git tags” that are “refs”

git HEAD Reference



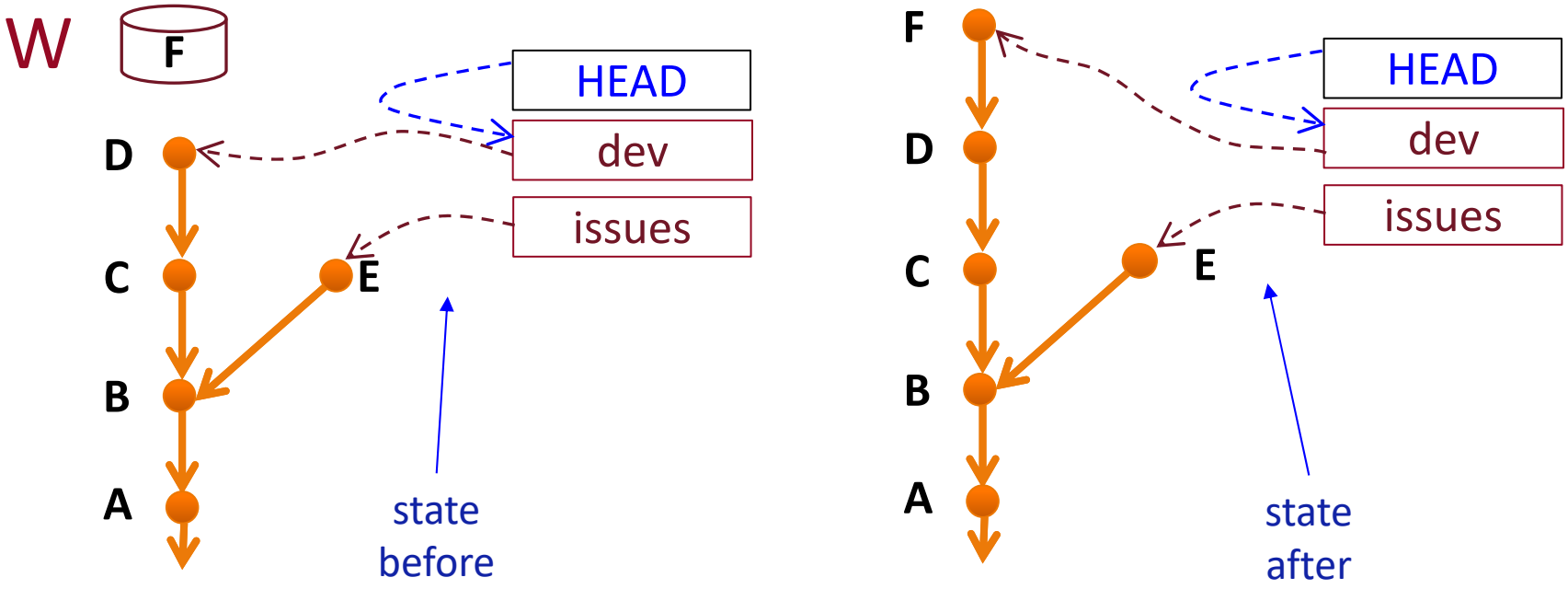
- Special reference HEAD
  - Commit/branch currently checked out
    - Workdir changes are considered w.r.t. HEAD
  - HEAD is an indirect reference
    - Points to another reference (the current branch)
    - E.g., above says that “dev” is the current branch

Commit the Workdir Changes



- Scenario:
  - You made changes in Workdir (now **F**)
  - Want to save these changes as a new commit

Commit the Workdir Changes



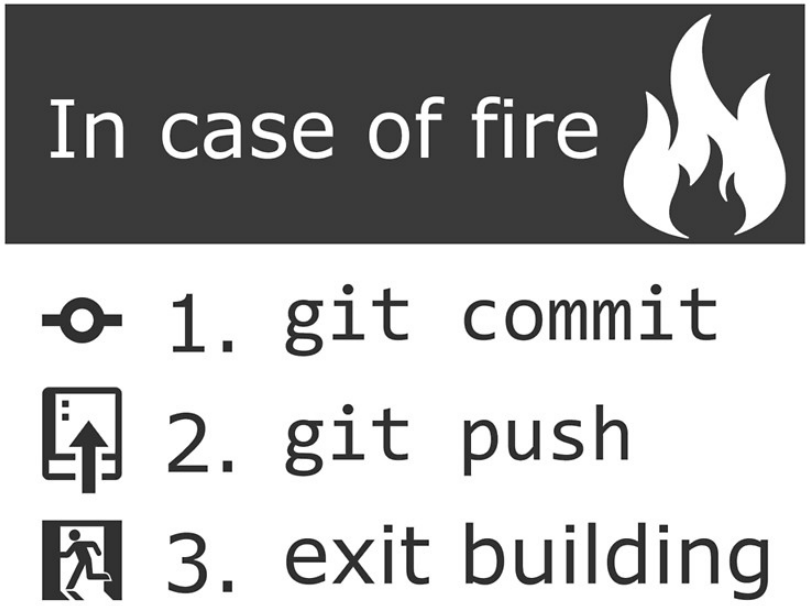

```
dev> git commit –m ”msg”
```

Current branch

- New commit based on changes in W from HEAD
  - Advance the branch pointed to by HEAD

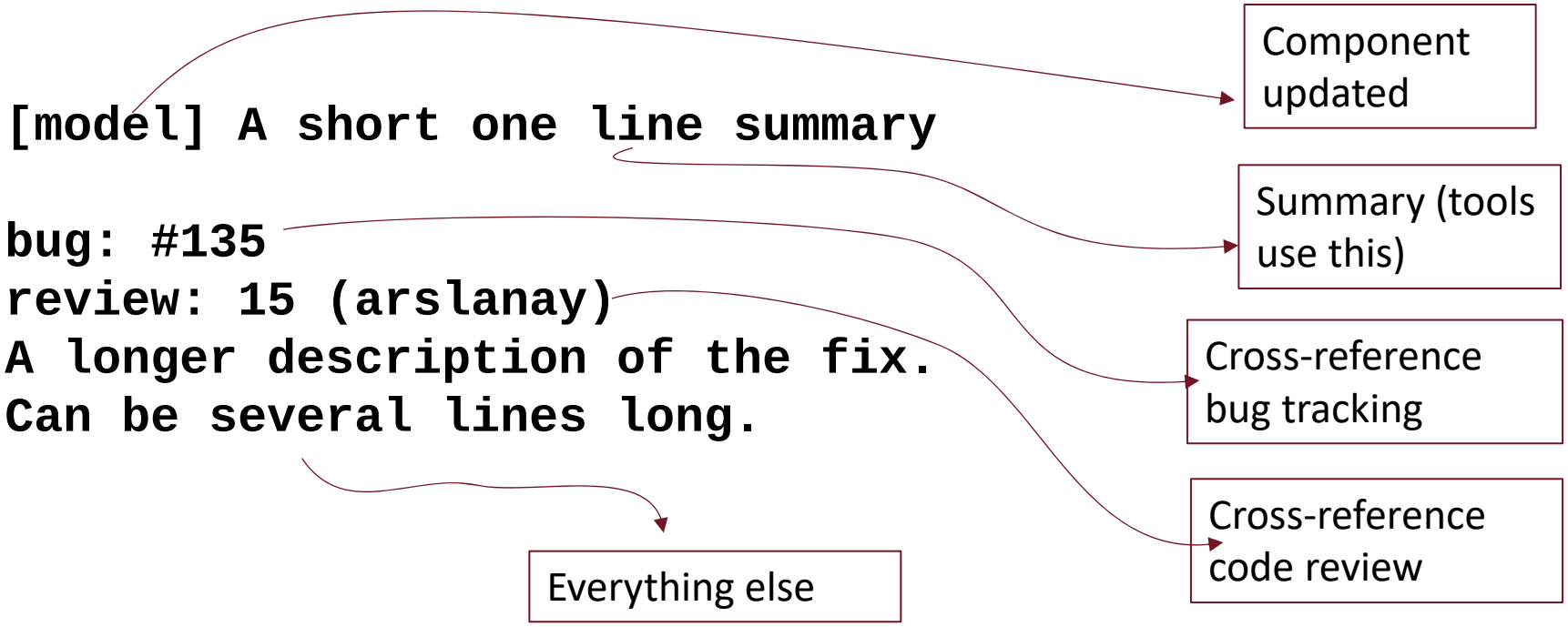
Good Practices

- Always provide a meaningful commit message.
- Commit often



Commit Messages

- Always use a meaningful commit message
- Example:



Commit – Good Practices

- Place every logically separate change into its commit
  - Allows more meaningful commit messages
  - Can be reverted independently
- Commit very often locally
  - E.g., after some tests pass before you change more
  - Commit even if a draft

    "If it's in the repo, it is safe"
- Later, we will re-package/cleanup the commits for sharing and archival

Add Files, Change Files, Commit, Push or Pull

- Warning!
- Never change both remote and local files without syncing them with a pull or push.
- If you do, you will not be able to pull or push until you resolve the conflict (which may not be easy).
  - If you change something local (i.e. on your computer), and you want to change something remote (i.e. on GitHub), make sure you do a push first to sync remote to local.
  - If you change something remote (i.e. on GitHub), and you want to change something local (i.e. on your computer), make sure you do a pull first to sync local to remote.

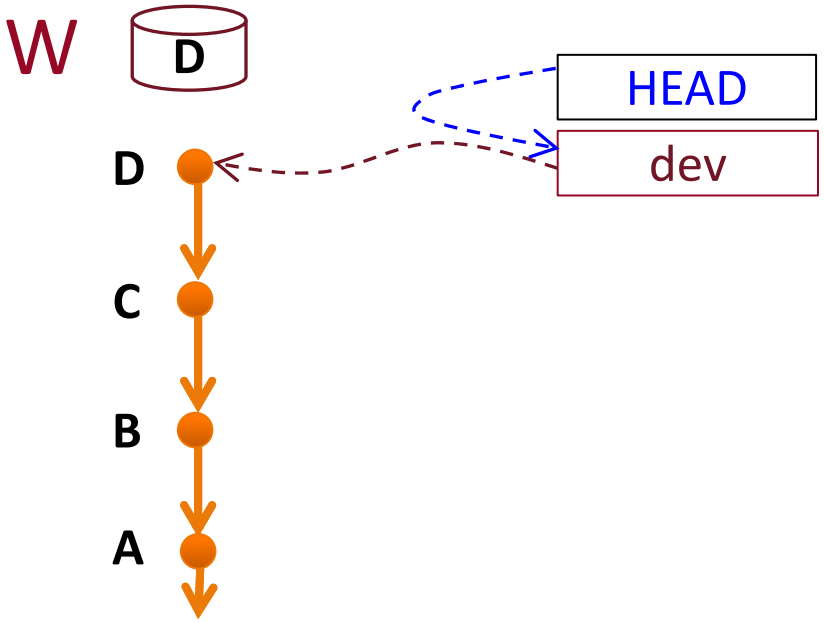
Git Command Summary

The following are some useful commands to remember at this point:

- `git init`
  - initializes version control. You do this once per project (i.e. assignment)
- `git branch`
  - will show you your branches and which one you are on
- `git remote add origin <repo-link>`
  - adds a remote repo to your local git
- `git add file1.ext file2.ext`
  - adds the files listed for staging to be committed
- `git commit -m "some message"`
  - will commit the staged files with "some message"
- `git push -u origin main`
  - will push changes to the remote repo
- `git pull origin main`
  - will pull changes to the local repo
- `git ls-files`
  - will list files being tracked
- `git status`
  - will tell you git's current status
- `git remote -v`
  - will tell you the remotes you have connected

Using Branches

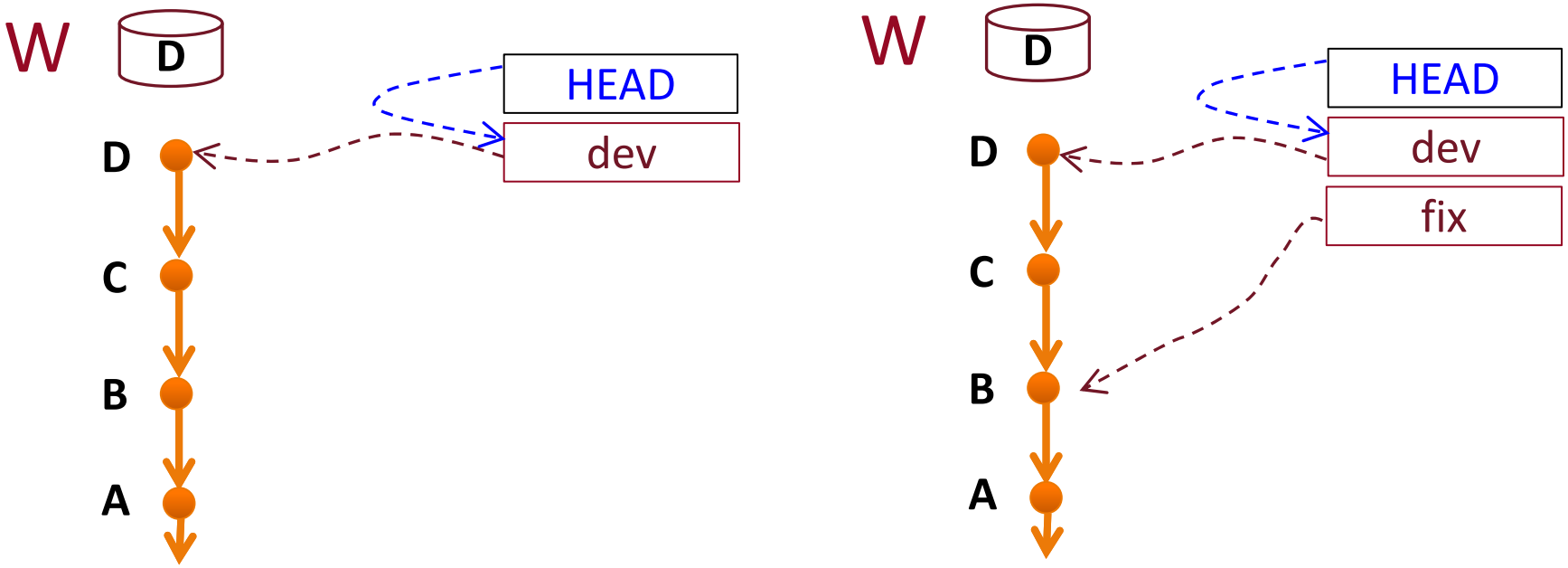
Creating Branches (git References)



Scenario:

- Want to create another reference for B
  - A more human-readable way to refer to B
  - Perhaps because we want to do some work on top of B

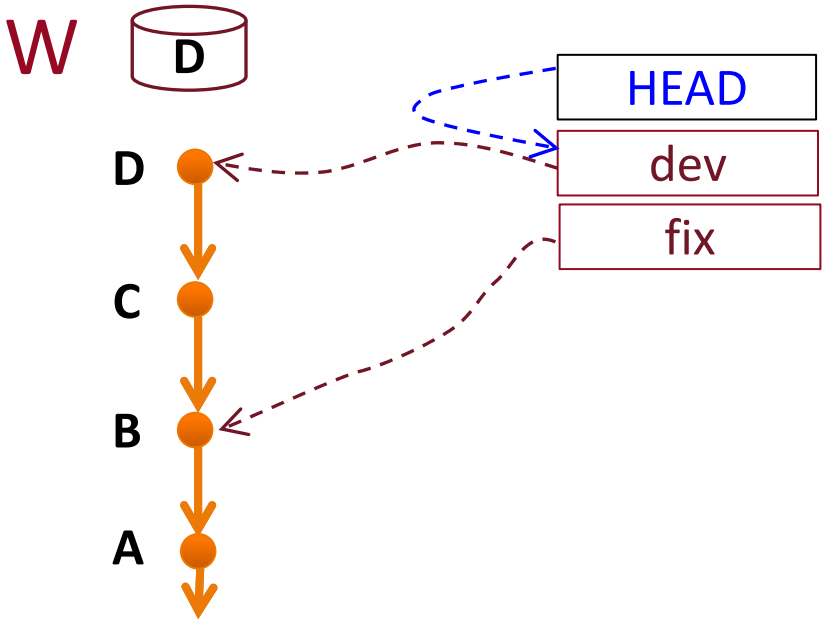
Creating Branches (git References)




```
dev> git branch fix B
```

- A branch starts initially as a reference
  - This does not change the current branch !
  - HEAD still points to “dev”
  - Workdir is not changed.

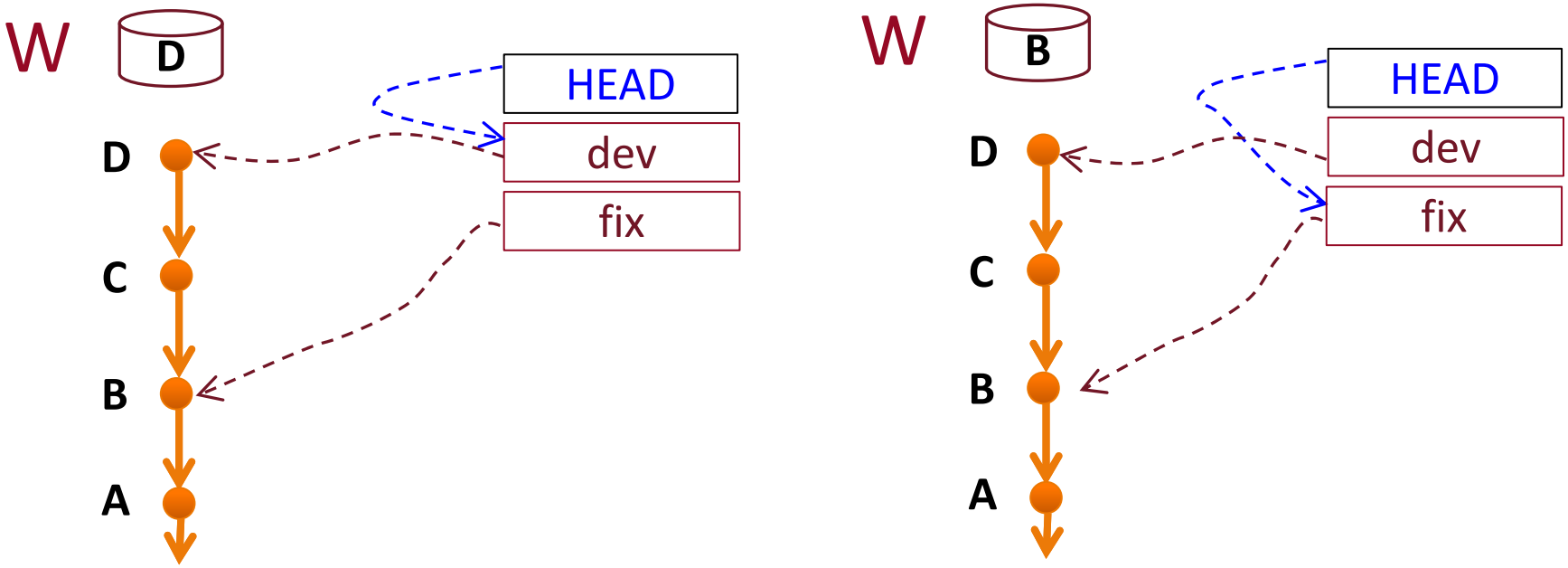
Check Out Another Branch



Scenario:

- Want to switch to work on top of “fix” (B)
  - Want to set Workdir to the contents of B
  - And set the current branch (HEAD) to “fix”

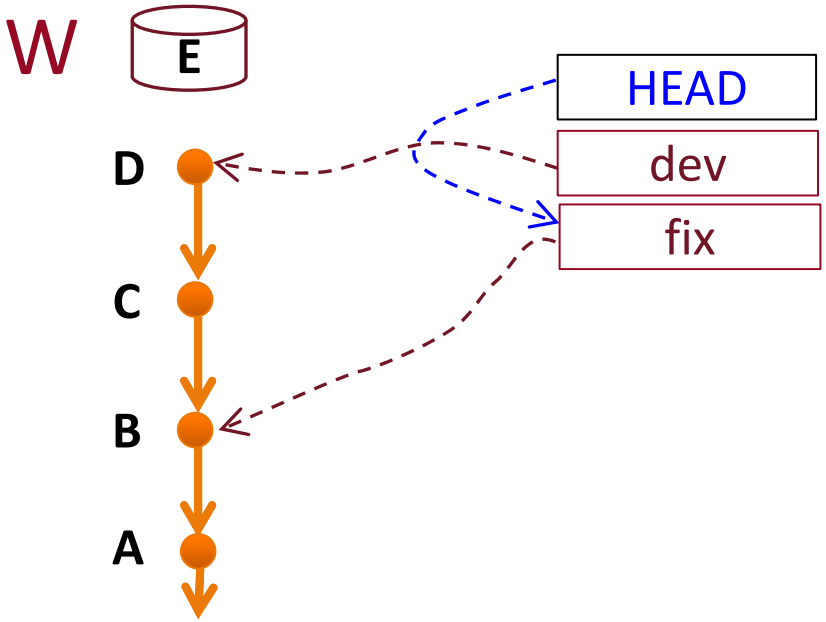
Check Out Another Branch




```
dev> git checkout fix
```

- Copy into Workdir a snapshot from repo, and set HEAD
  - HEAD points to the new branch now
  - Working directory changes !!
  - This is how you switch to work on another branch

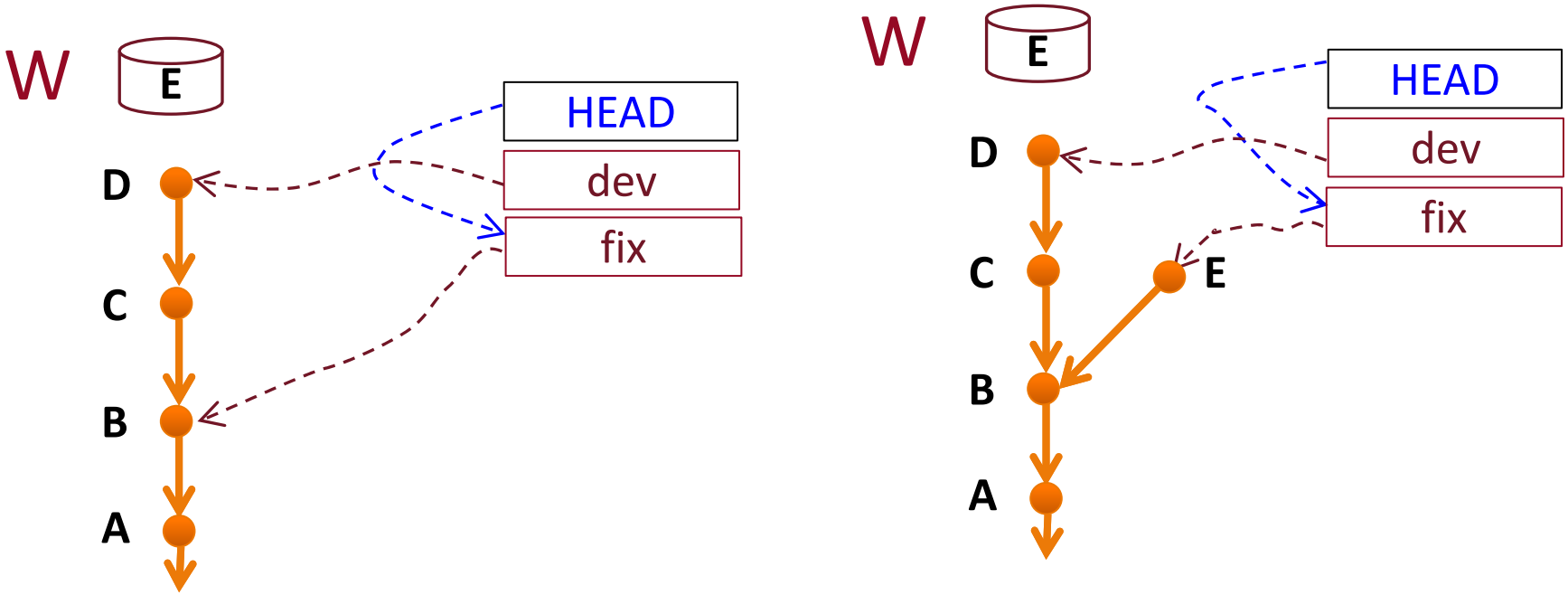
Add Commits to a Branch



Scenario:

- You had switched to work on branch “fix”
- And you made changes in Workdir (E)
- Want to commit those changes
  - B will be the parent

Add Commits to a Branch



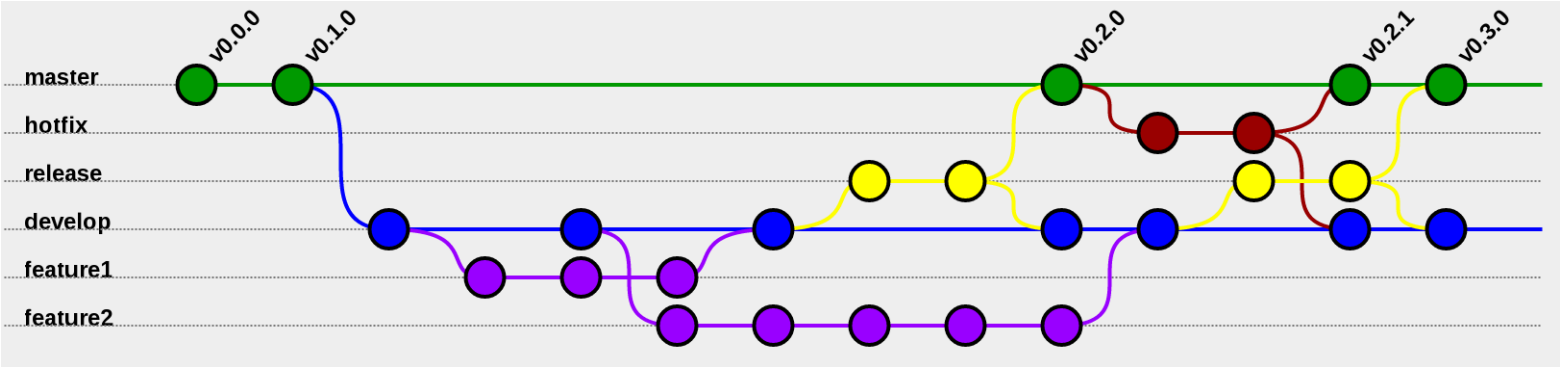

```
fix>  git commit -am "…"
```

- A commit is added to the current branch
  - The current branch "fix" is advanced
  - You achieve a branching structure by making more than one commit on top of the same parent commit
    - e.g., C and E both on top of B

Uses for Branches

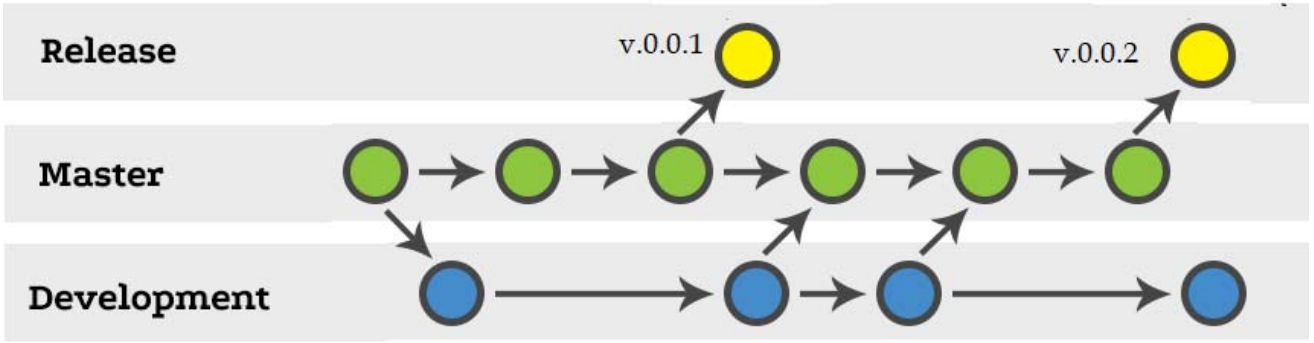
- Separate branch for custom release
- Need to fix a bug on a previous version and rerelease that version
- Snapshot of code for testing
  - Development continues on main trunk
  - Testing and hot-fixes on testing branch
  - Eventually all hot-fixes merged to trunk
- Temporary (or private) versions
  - For implementation of new features
  - Isolates changes
  - Eventually merged back into common branch (trunk)

Branching Strategies (I)



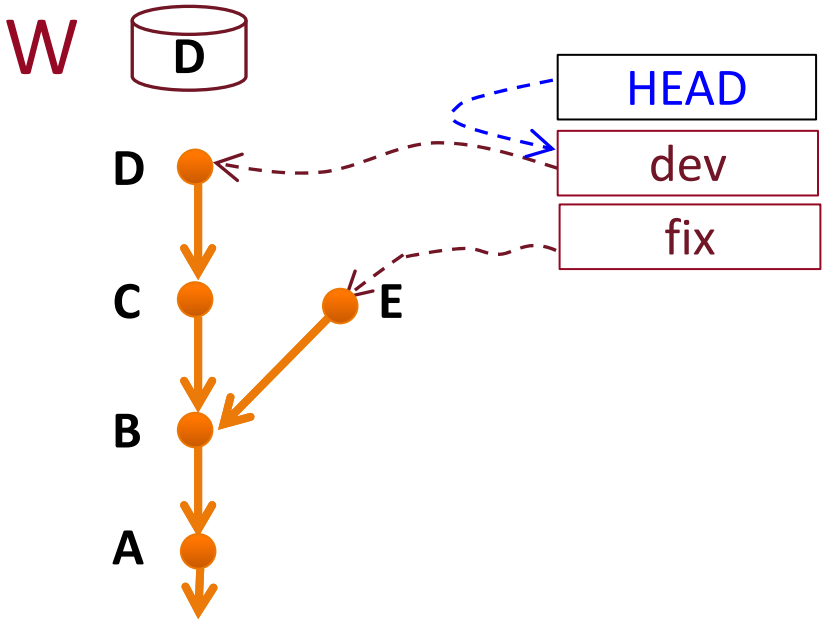
- Stable common branch (trunk): Use branches for small-team development and keep in trunk only stable releases
- Advantages:
  - Trunk is always stable, little interference between developers and between teams
- Disadvantages:
  - Delays integration, huge merge at integration time
  - Each big merge is an opportunity to make mistakes
  - Somebody might have to merge conflicting changes made by others
  - Don’t try the big-bang merge too late in the iteration !

Branching Strategies (II)



- Develop in common branch (trunk): Make branches for releases
- Advantages:
  - "continuous integration", problems surface early
  - But trunk may be in unstable state at times
- This scheme often works well in small and medium size team
  - Automatic testing is best

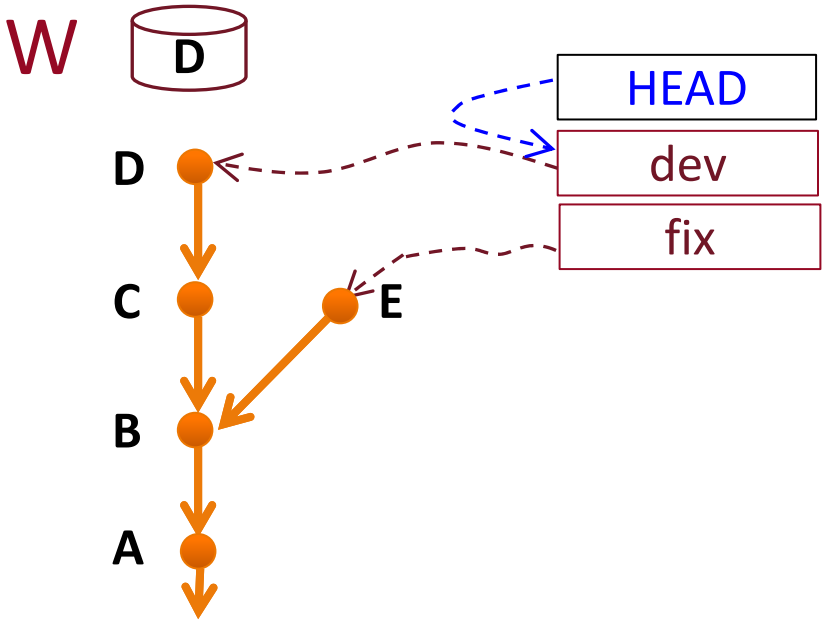
Merging Branches

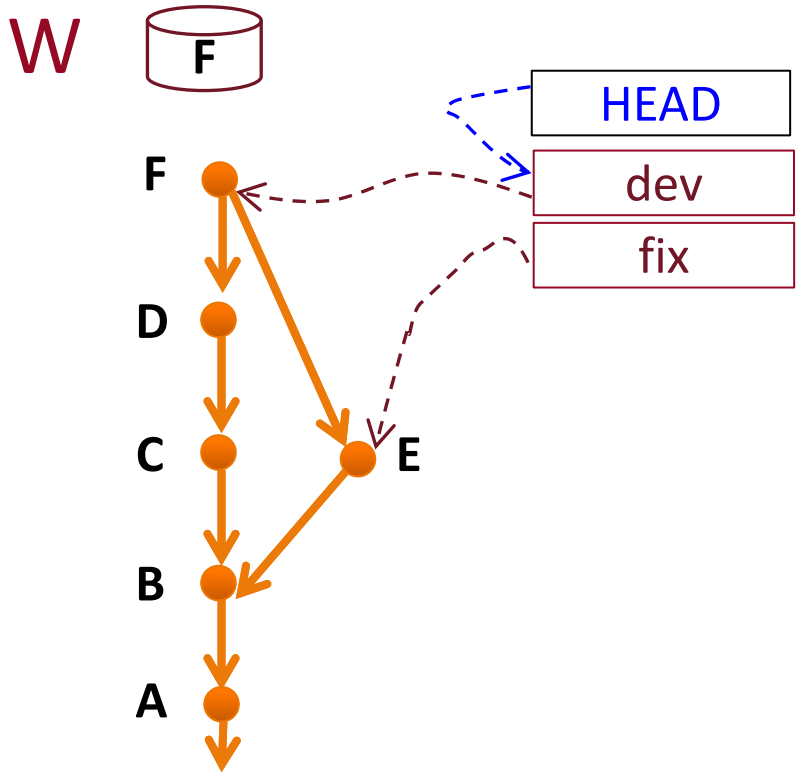


Scenario:

- You are on “dev” at D (which you committed)
- A colleague made changes (E) based on B
- Want to incorporate those changes into your work

Merging Branches

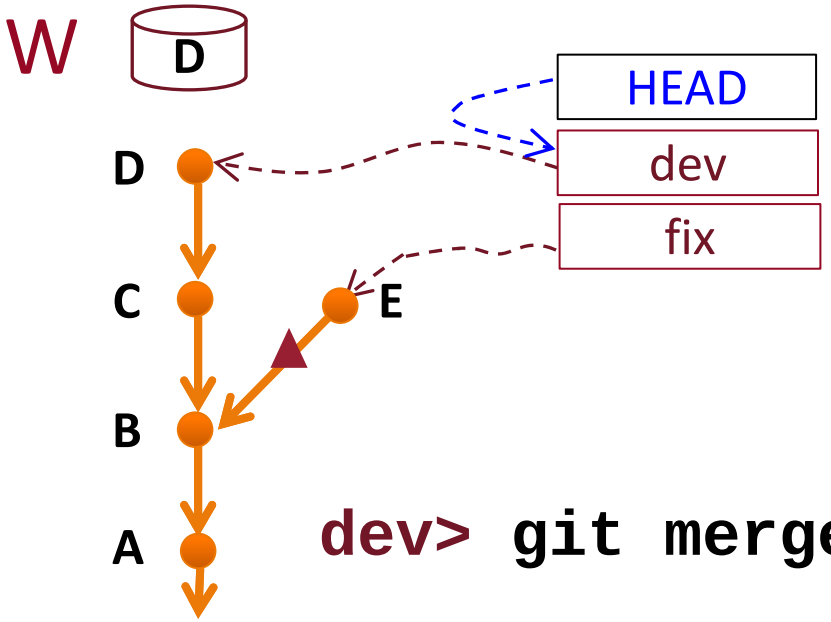




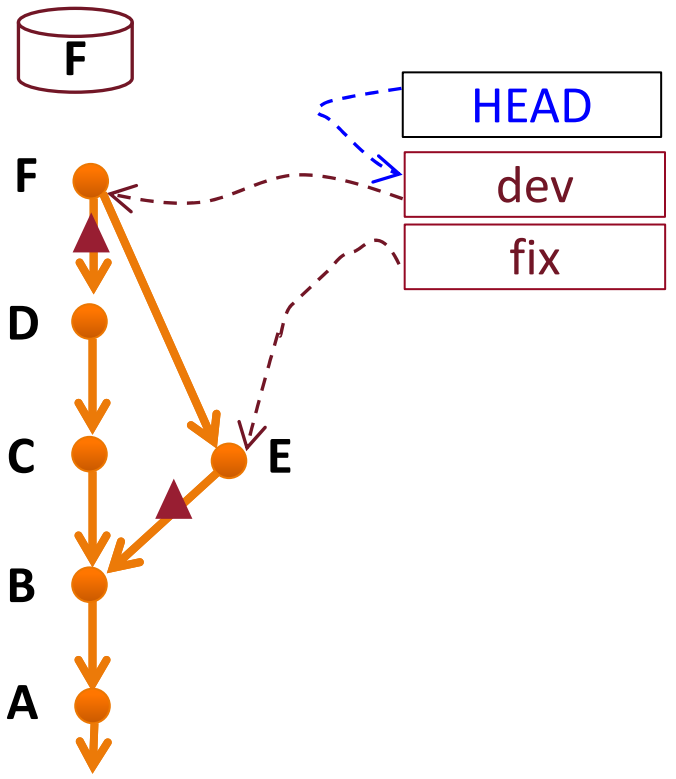

```
dev> git merge fix
```

- Creates a new commit in current branch
  - Includes changes B-C-D and also B-E
  - Working directory changes, current branch is advanced

Anatomy of a Merge




```
dev> git merge fix
```




```
B :
20: void f(int i){
21:   int j = 2;
22:   print(i+j);
23: }
```

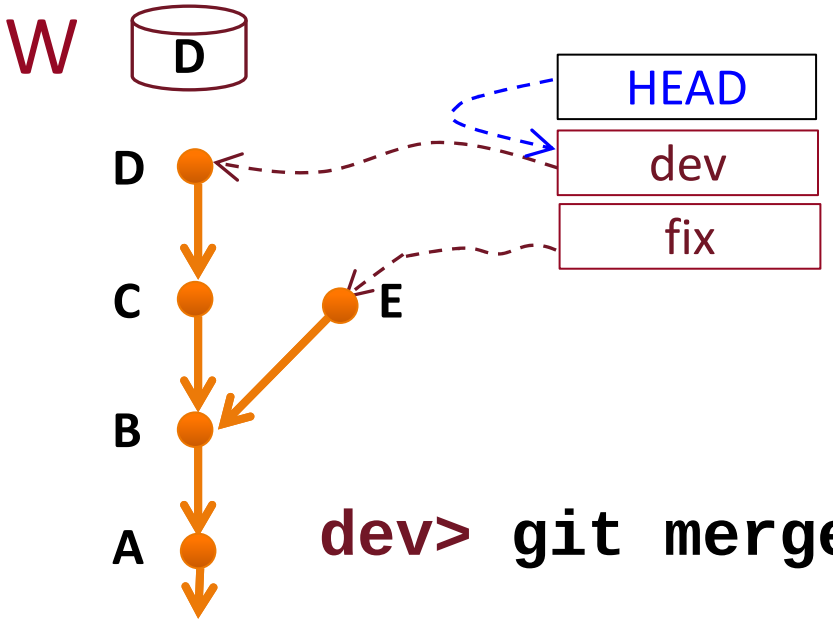
```
E :
20: void f(int i){
21:   int j = 3;
22:   print(i+j);
23: }
```

```
D :
20: void g(int i){
21:   int j = 2;
22:   print(i+j);
23: }
```

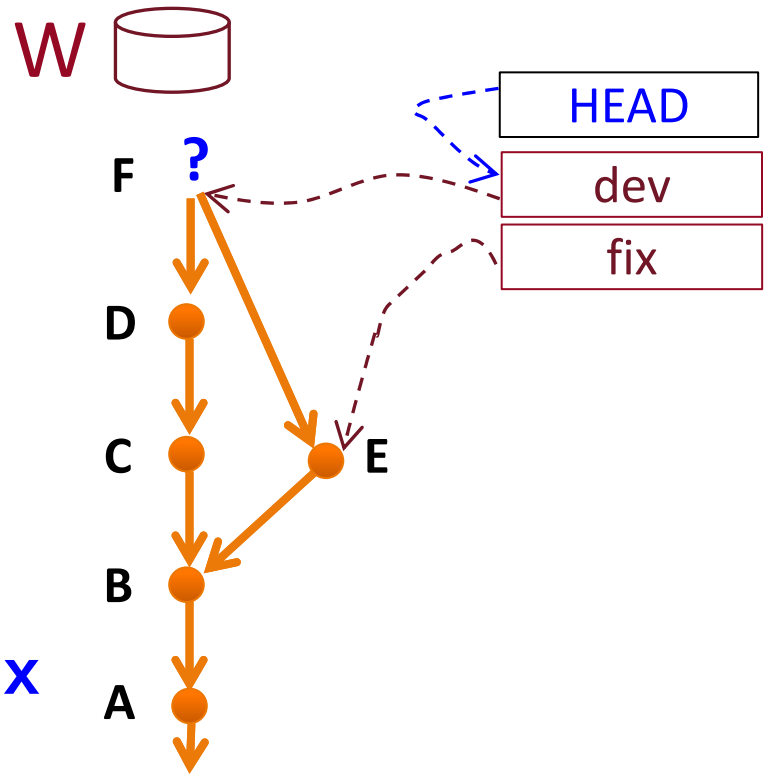
```
F :
20: void g(int i){
21:   int j = 3;
22:   print(i+j);
23: }
```

- Find common ancestor of current branch “dev” and “fix” (B)
- Compute changes from ancestor to E (fix)
  - B-E: “replace line 21 with int j = 3;“
- Apply these changes to D (dev). Obtain F

Merge Conflicts



dev> git merge fix




```
B :
20: void f(int i){
21:   int j = 2;
22:   print(i+j);
23: }
```

```
D :
20: void g(int i){
21:   int j = 4;
22:   print(i+j);
23: }
```

```
E :
20: void f(int i){
21:   int j = 3;
22:   print(i+j);
23: }
```

```
F :
20: void g(int i){
21:   int j = ??;
22:   print(i+j);
23: }
```

- Combining changes B-C-D + B-E yields 21: int j = 3;
- Combining changes B-E + B-C-D yields 21: int j = 4;
- Merge conflict, not safe, git aborts, human intervention needed

Merge Conflicts

- When the changes to be merged yield different results depending on the order they are applied, we have a merge conflict !
- Merge leaves partially merged file:

```
void f(int i) {
 <<<<<<<< HEAD
int j = 3;
 ========
int j = 4;
 >>>>>>>> fix
```

Highly recommended: use kdiff3 or another “3-way graphical merge ” tool

GitHub Pull Requests

- A GitHub Pull Request is a *request* to *merge* one branch into another.
  - https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request
  - Same as “merge request” in GitLab
- Pull request can be used to interchange the code between other people and discuss the changes with them easily.
- How to create a GitHub pull request?
  - Demo in class.
  - Here is a blog post that walks you through the same process.
    - https://www.better.dev/create-your-first-github-pull-request

GitHub Pull Requests

```
git branch my-new-branch
git checkout my-new-branch
# Make the changes in code.
git add <changed files>
git commit -m "My commit message"
git push origin my-new-branch
# Open the merge request link (can also open it from GitHub)
# Choose the branch to merge to; submit merge request.
```

```
# Make sure to fetch/pull the merged commit to the `main` on all
local repos connected to
git checkout main
git pull origin main

#if you would like to delete the pull request branch
# delete remote branch
git push origin --delete my-new-branch
# delete local branch
git branch -D my-new-branch
```