

CptS 322- Programming Language Design

Version Control and git Basics

Instructor: Sakire Arslan Ay
Fall 2021

Outline

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

Version Control

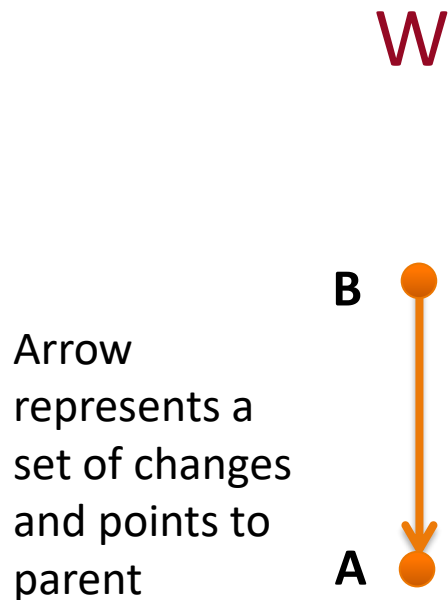
- VC tracks multiple versions of your project
 - Different releases
 - Different versions of the same release
 - In an extreme, every time you edit the project

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

- 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



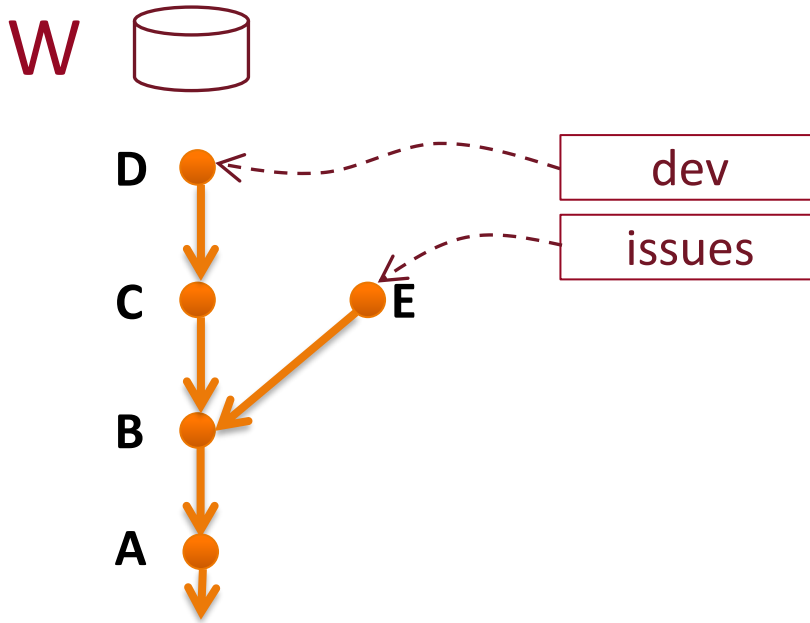
git Revision Names



Arrow
represents a
set of changes
and points to
parent

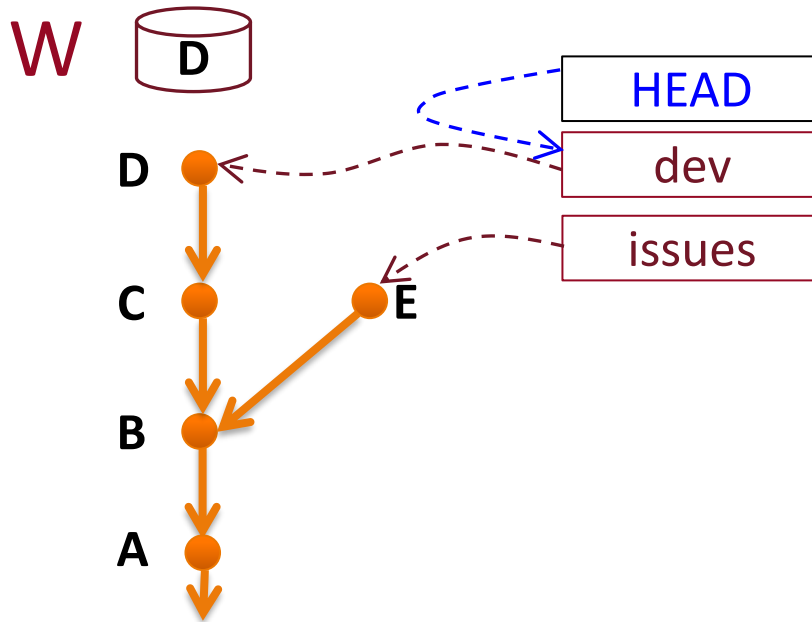
- git names revisions with a 40-char hexadecimal string
- Typically first few (5) chars are unique and enough for commands

git References



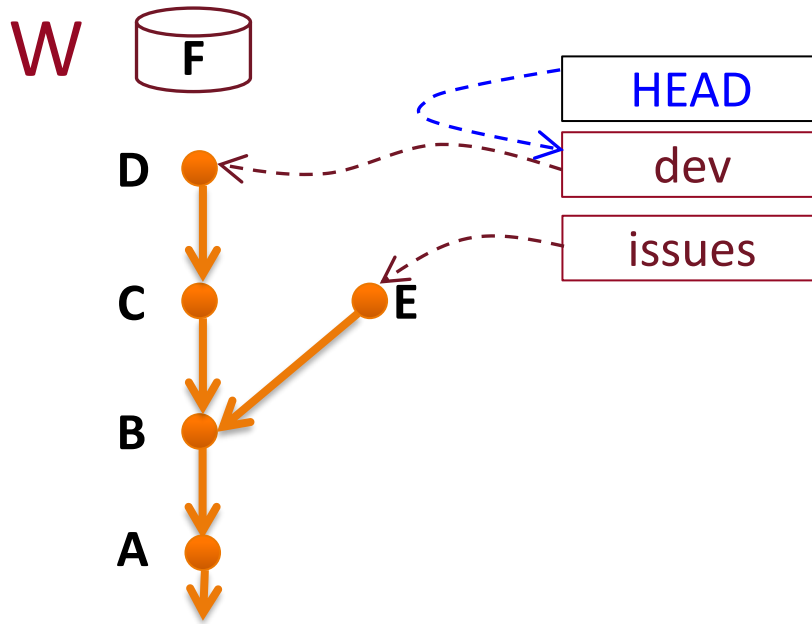
- git maintains a table of “references” (a.k.a. “refs”) which refer to commits in the repo
 - Branches are represented as “refs” (e.g., dev, fix)
 - There are also “git tags” that are “refs”

git HEAD Reference



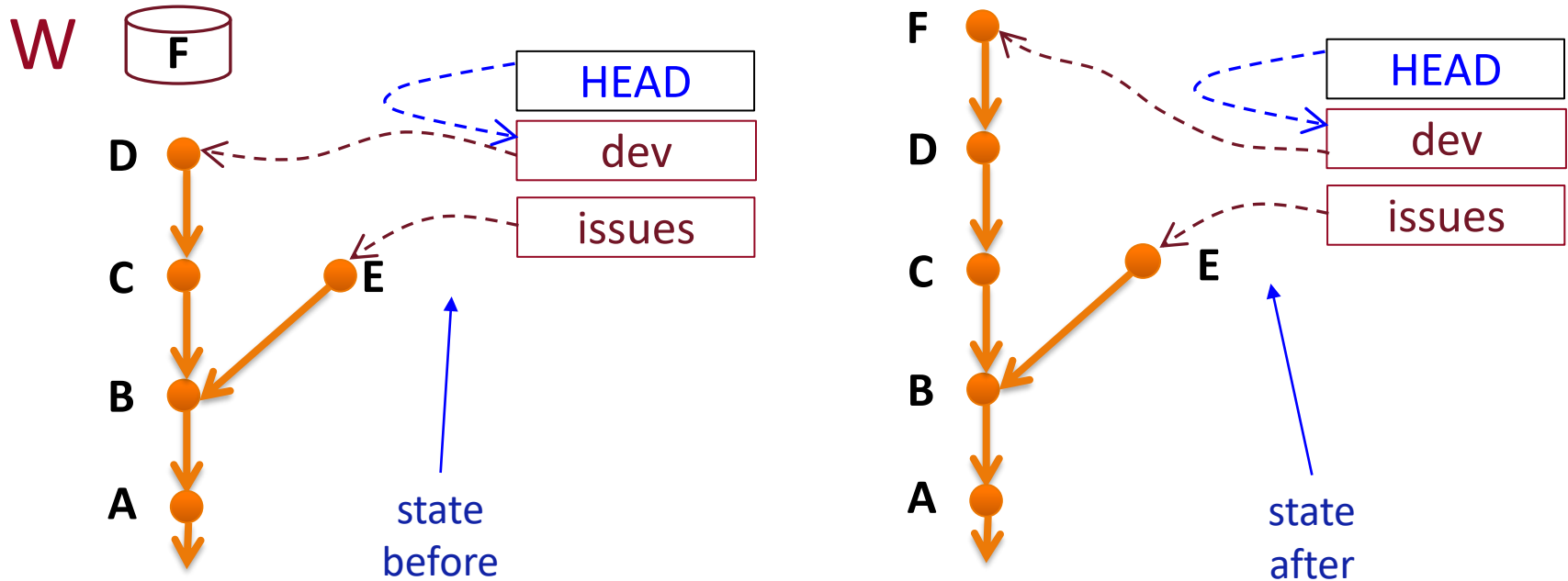
- Special reference **HEAD**
 - Commit/branch currently checked out
 - Workdir changes are considered w.r.t. **HEAD**
 - HEAD is an indirect reference
 - Points to another reference (the current branch)
 - E.g., above says that “dev” is the current branch

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



Current branch → **dev**> `git commit -m "msg"`




- 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

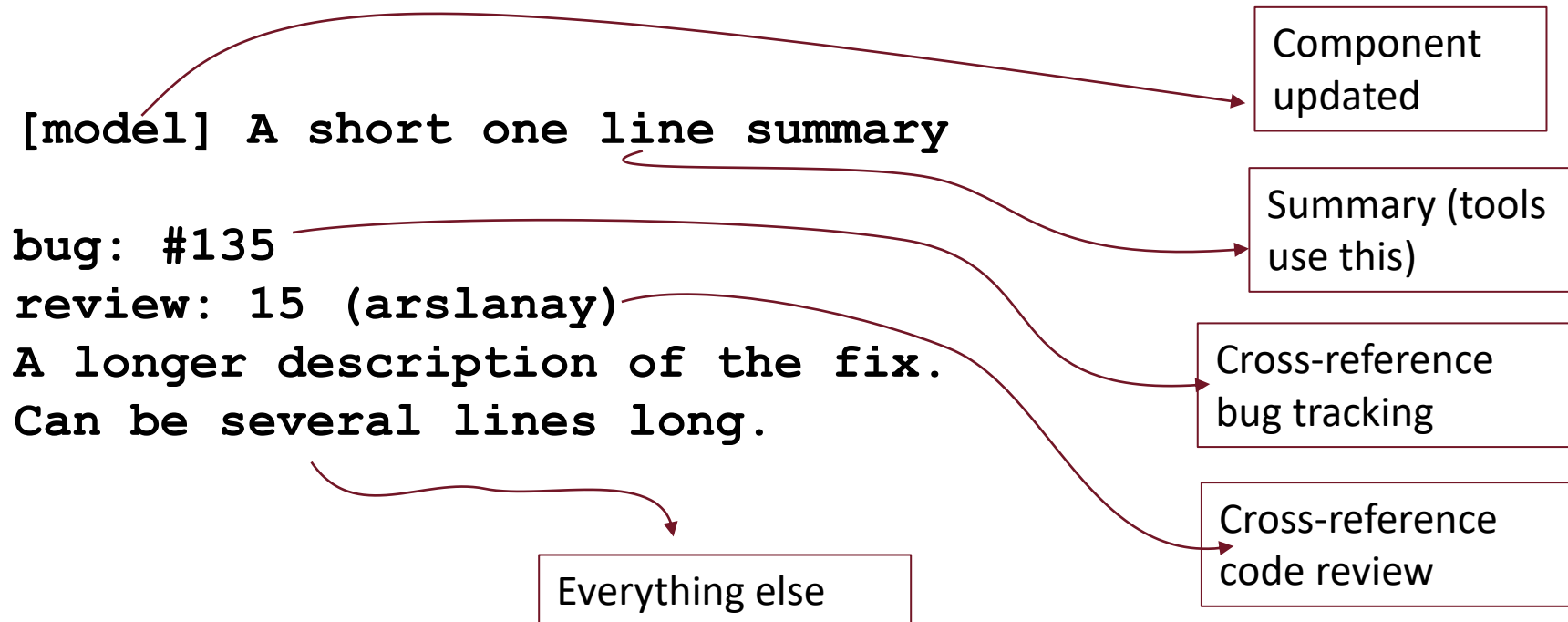
In case of fire



1.  git commit
2.  git push
3.  exit building

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.

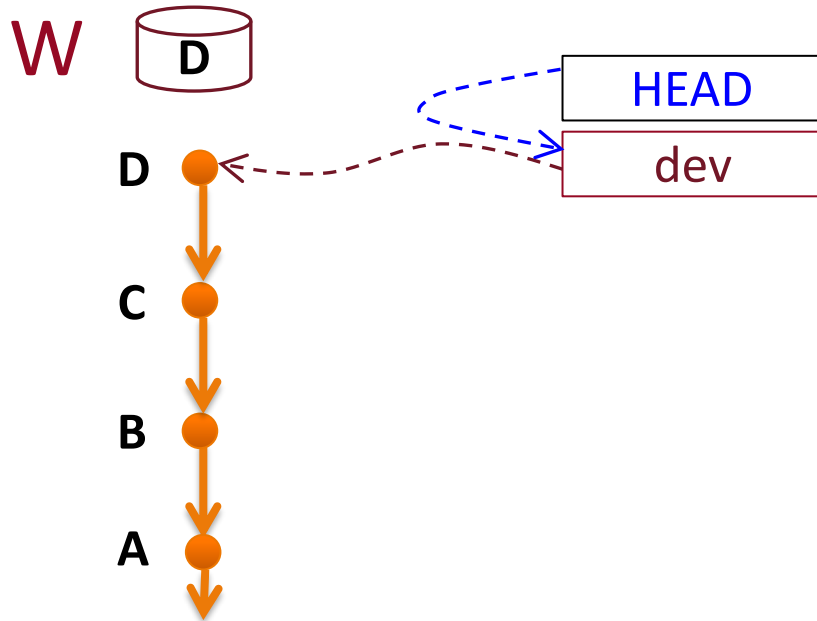
Git Command Summary

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

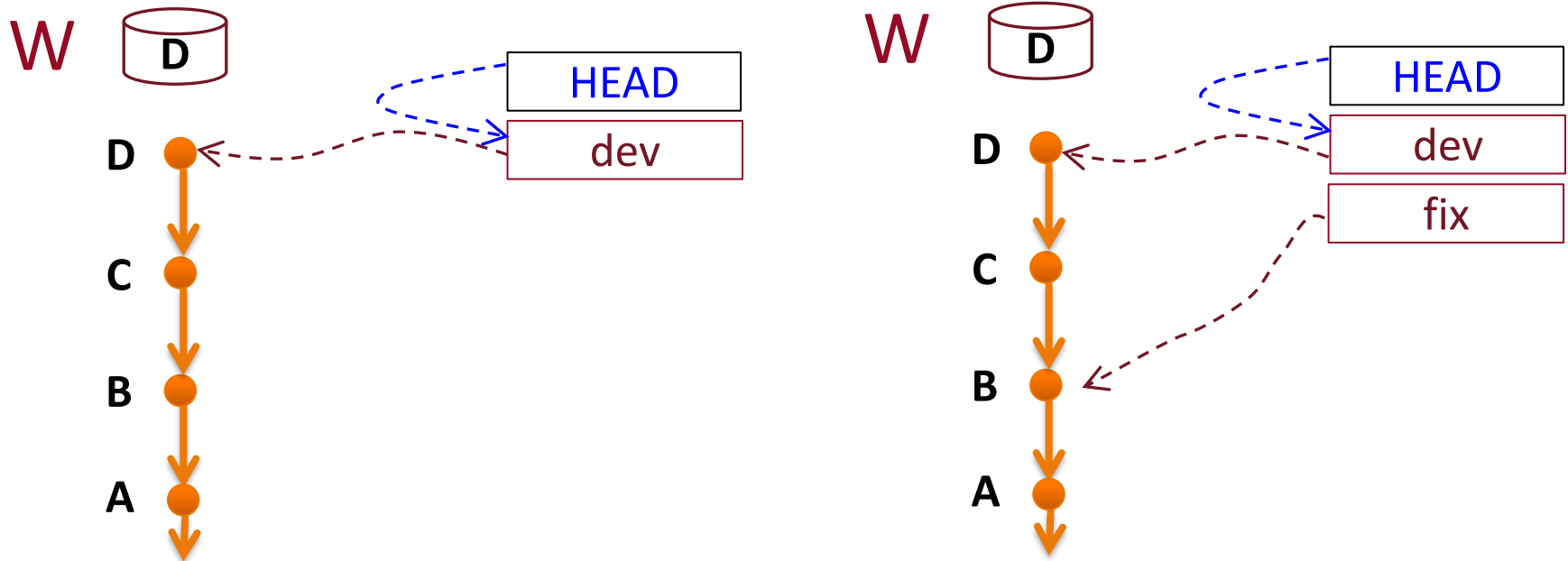
Creating Branches (git References)



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

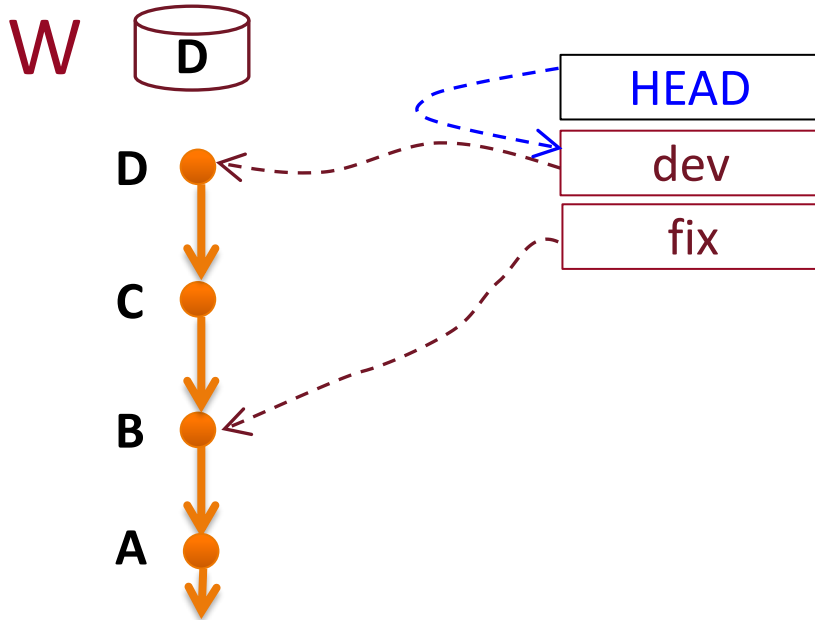
Creating Branches (git References)



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.

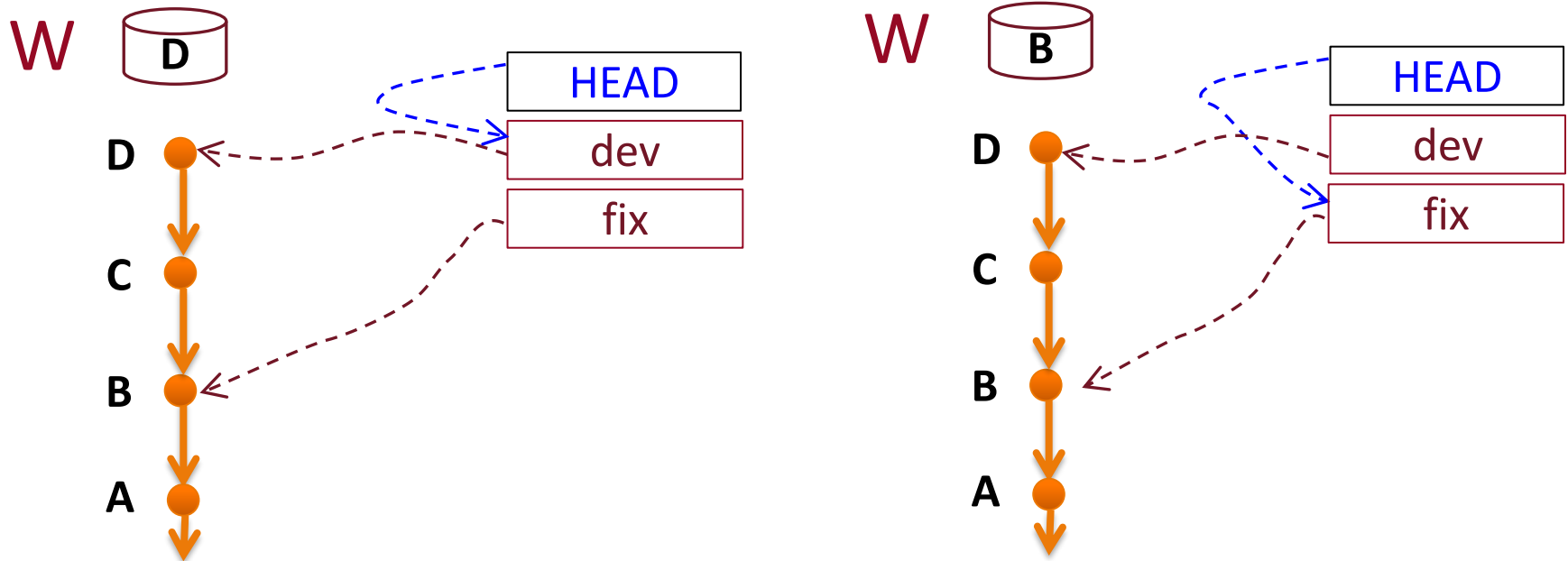
Check Out Another Branch



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”

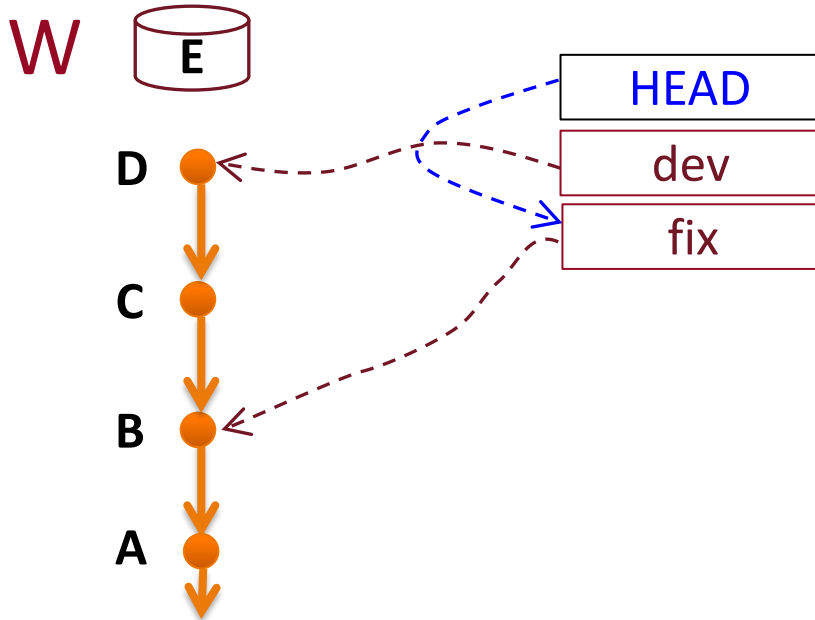
Check Out Another Branch



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

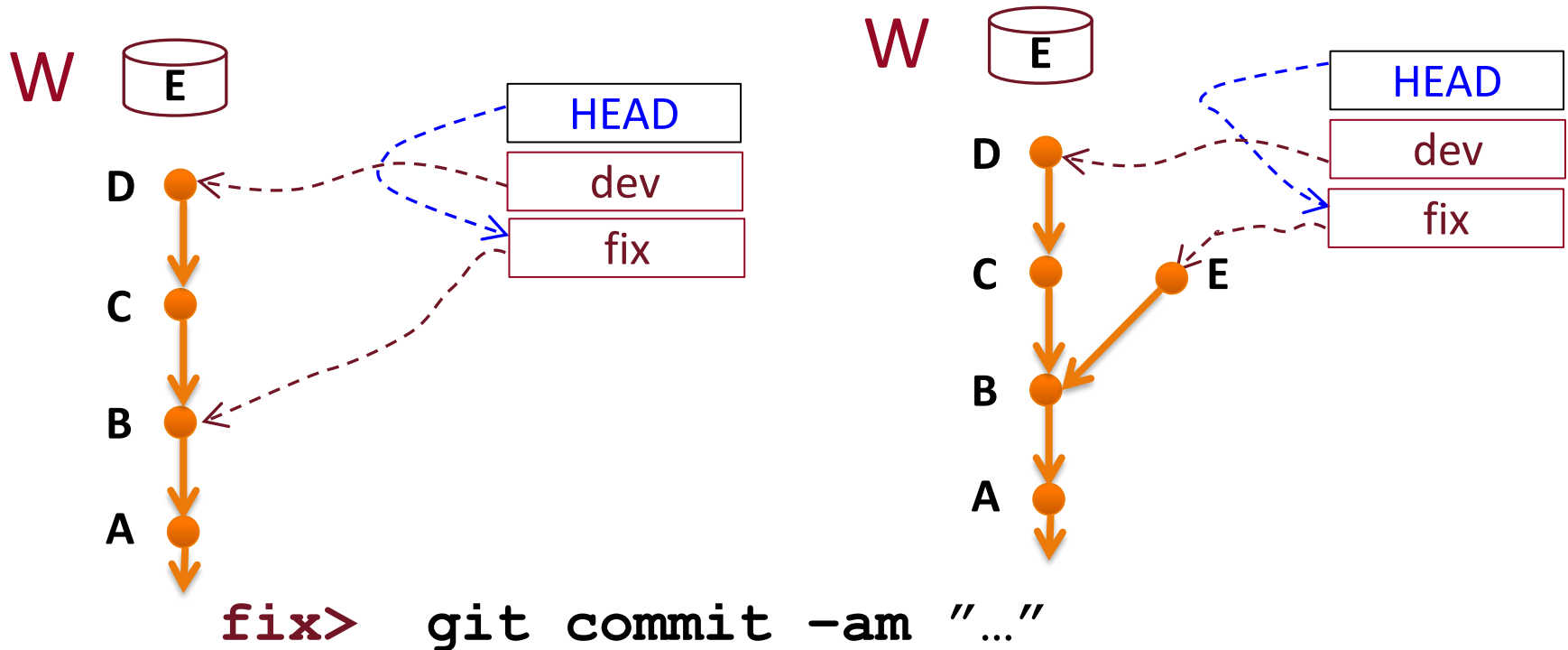
Add Commits to a Branch



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

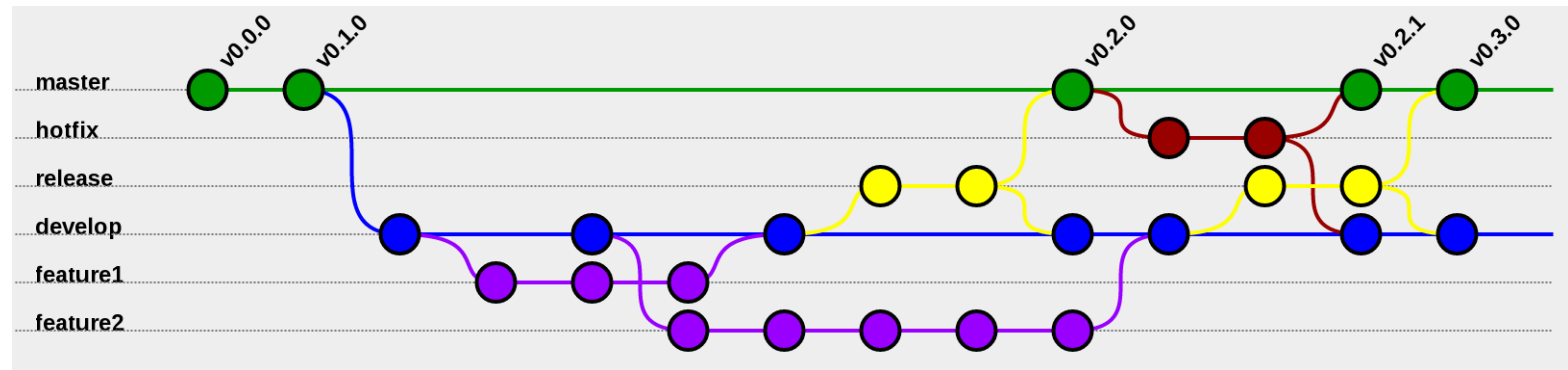


- A commit is added to the current branch
 - The current branch “fix” is advanced
 - You achieve a branching structure by making more than one commit on top of the same parent commit
 - e.g., C and E both on top of B

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)



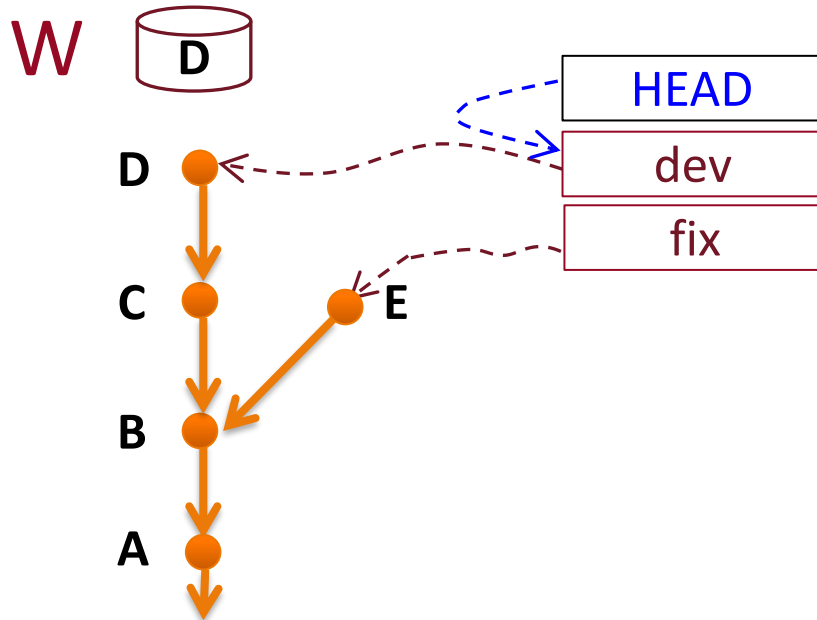
- Stable common branch (trunk): Use branches for small-team development and keep in trunk only stable releases
- Advantages:
 - Trunk is always stable, little interference between developers and between teams
- Disadvantages:
 - Delays integration, huge merge at integration time
 - Each big merge is 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

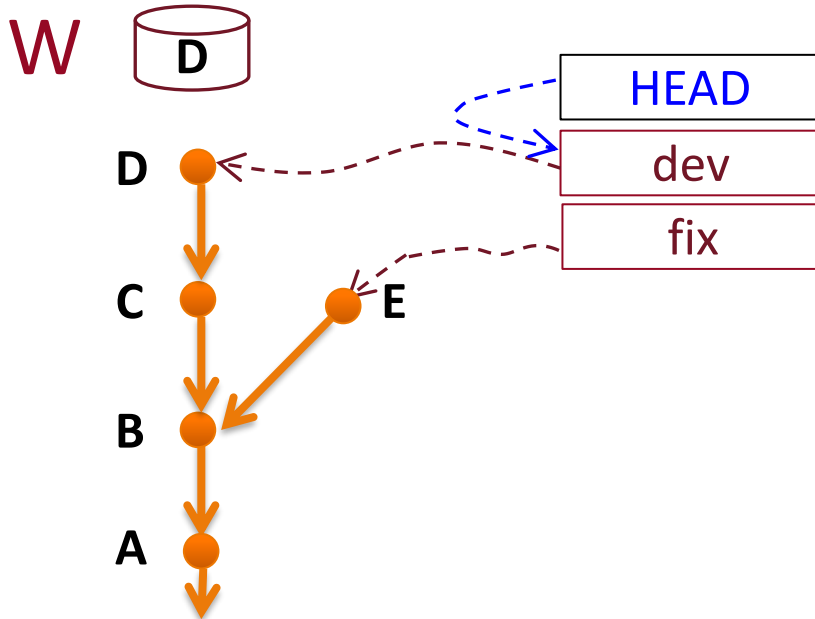
Merging Branches



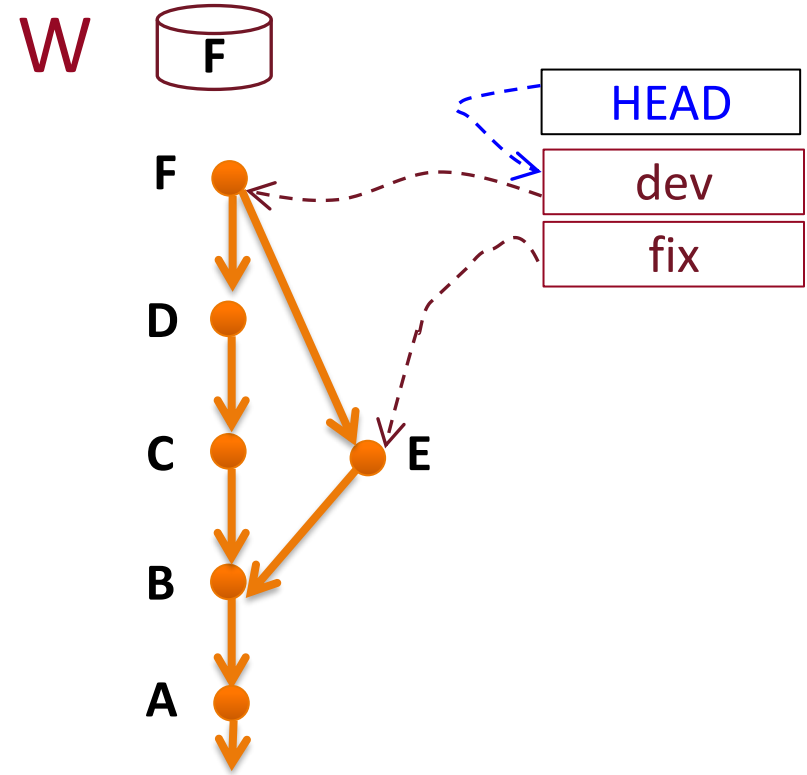
Scenario:

- 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

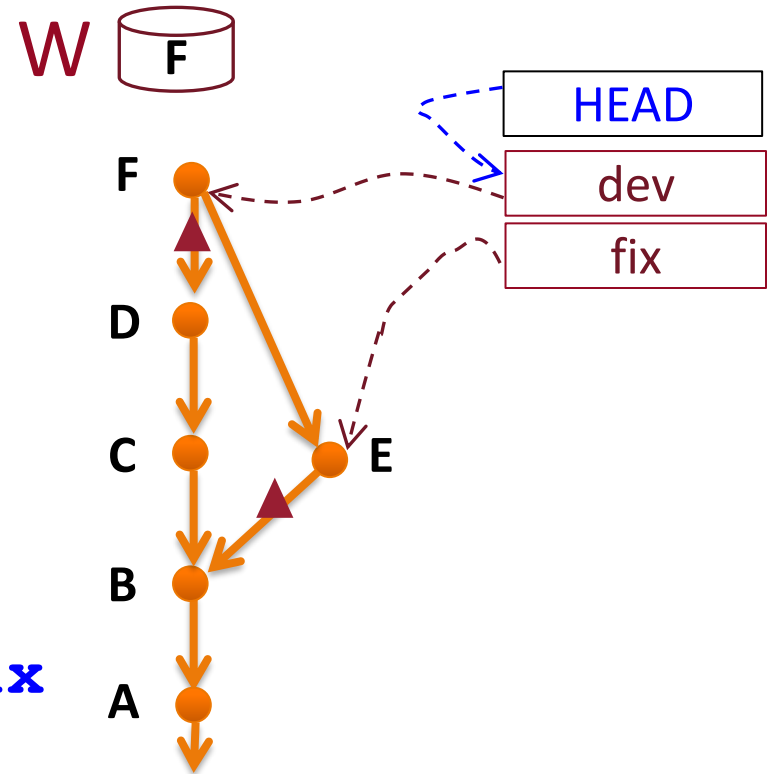
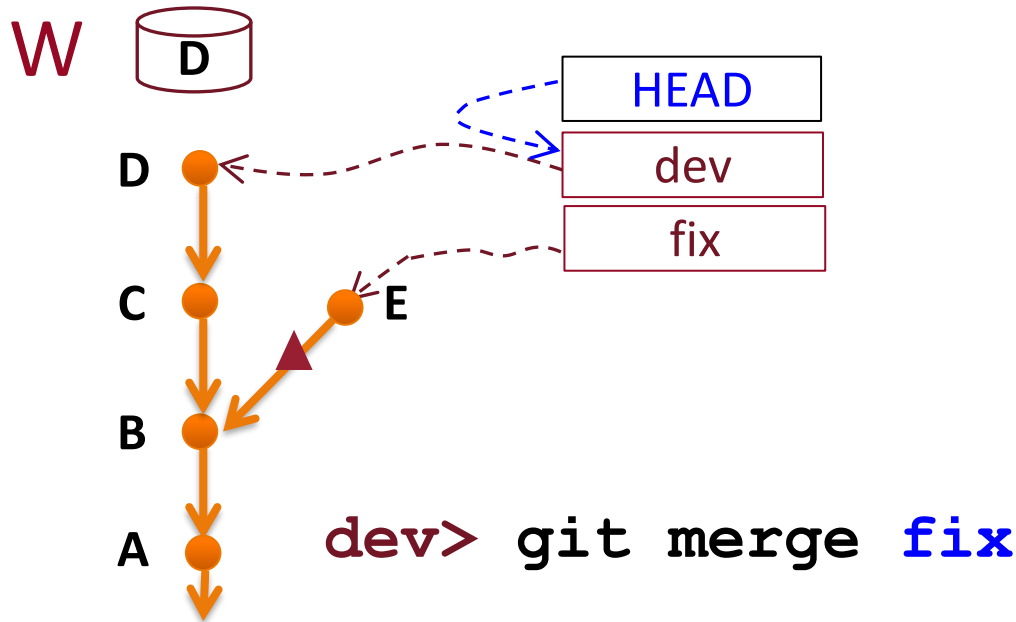


`dev> git merge fix`



- 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



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 :

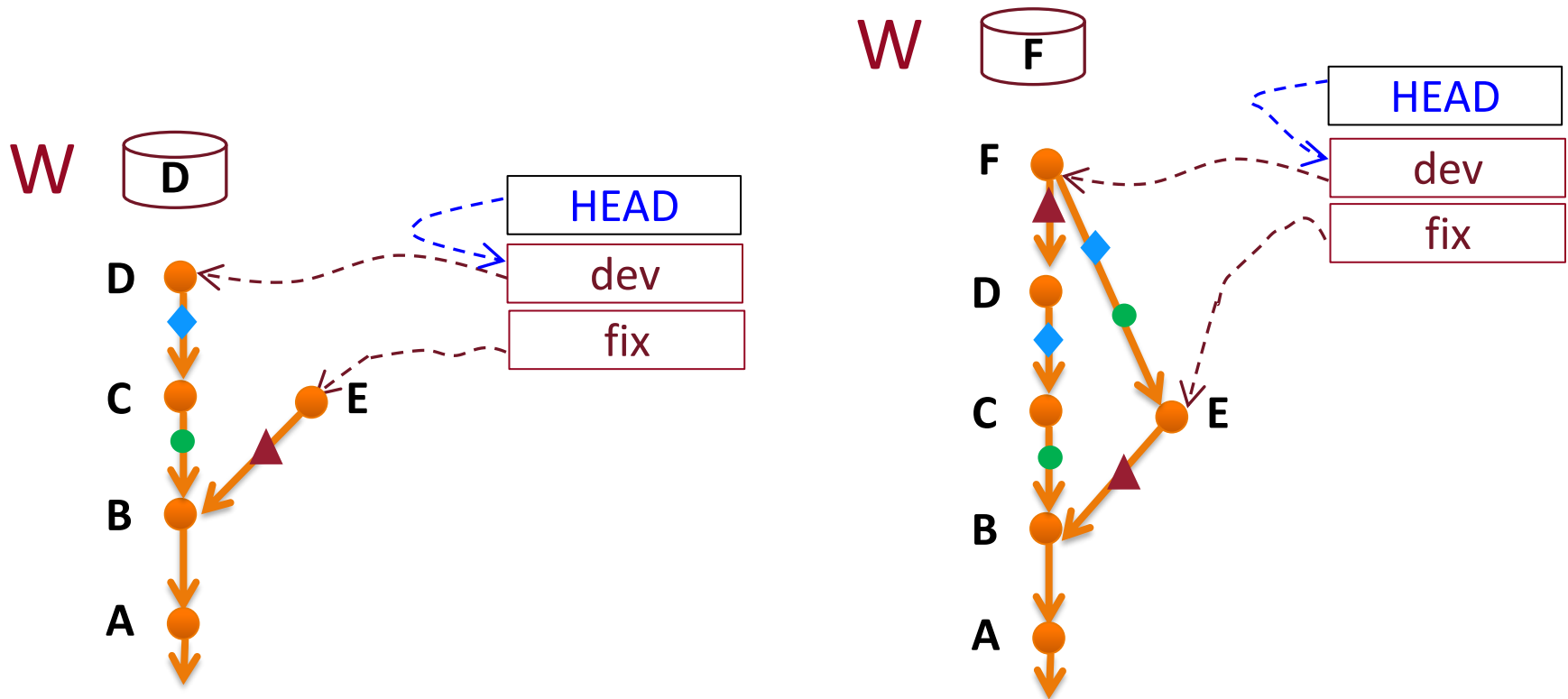
```
20: void g(int i){
21:     int j = 3;
22:     print(i+j);
23: }
```

- 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

Well-Defined Merge

- Merging involves combining two sets of changes from a common ancestor
 - A merge is well-defined if the result is the same no matter in what order we apply the two sets of changes

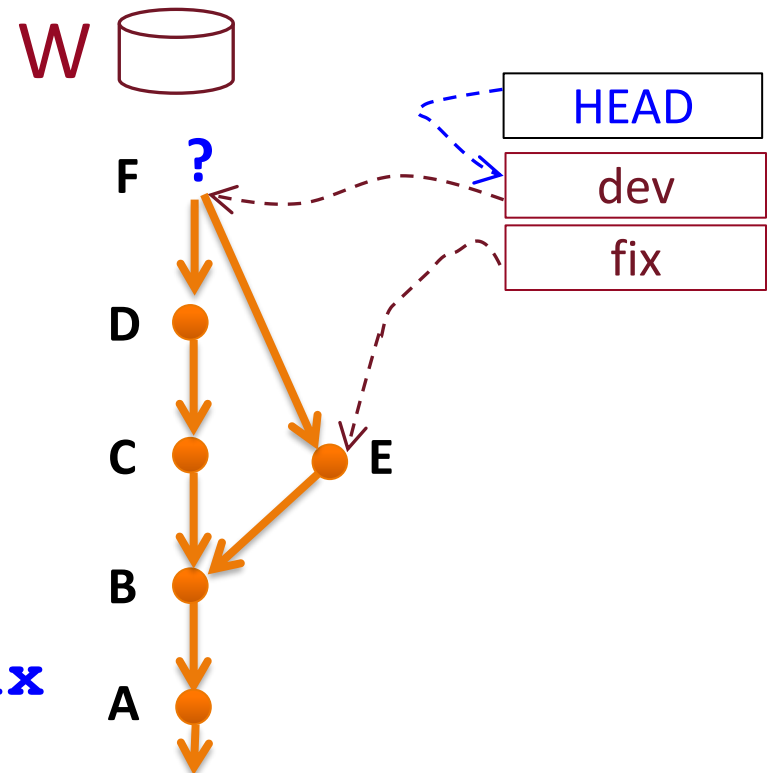
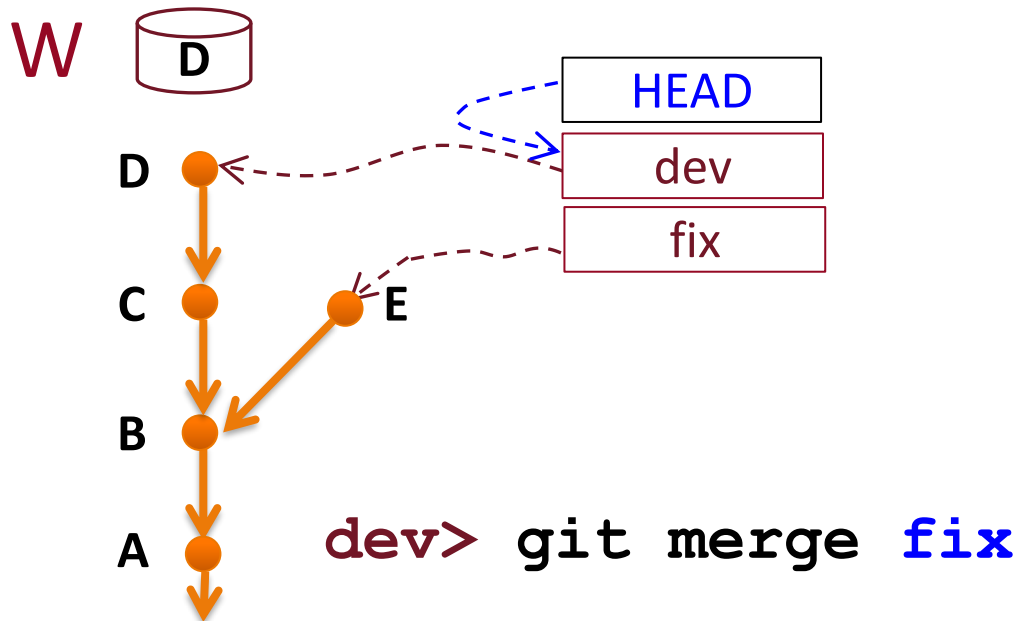
Well-Defined Merge - Example



dev> git merge fix

- Changes are commutative
 - $E-F = B-C-D$ ◆ ●
 - $D-F = B-E$ ▲
- — On all paths from F to B the set of changes is the same

Merge Conflicts



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

Resolving Merge Conflicts

- You can keep one of the two copies:
 - Edit the conflict file manually
 - Even better: graphical merge tools:
 - `git mergetool`
- Git forces you to commit your W before merge
 - Easy to undo the merge, no way to loose your work
 - Can always give up if you do not like the merge:
 - `git merge -abort`

Important: Conflicts are Syntactic

- Conflict detection is based on “nearness” of changes
 - Changes to the same line will conflict
 - Changes to adjacent lines will likely conflict
 - Changes to far-away lines will be considered to not conflict
- Note: Lack of conflicts does not mean the merged changes work together

Example With No (Syntactic) Conflict

- Ancestor:

```
int f(int a, int b) { ... }
```

- Branch “refactor”:

```
int f(int a, int b, int c) { ... }
```

 - And also add argument “c” to all calls to “f”
- Branch “dev” based on ancestor:
 - Add another call to f (with two arguments)
- Merged program
 - Has no merge conflicts
 - But will not even compile

Important

- Merging is syntactic
- Semantic errors may not create conflicts
 - But the code is still wrong
 - You are lucky if the code fails to compile
 - Worse if it does compile in spite of a semantic conflict . . .
 - E.g., when you use a language without a static type checker
 - To avoid problems:
 - Maintain good team communication
 - Even better, have good automated tests

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

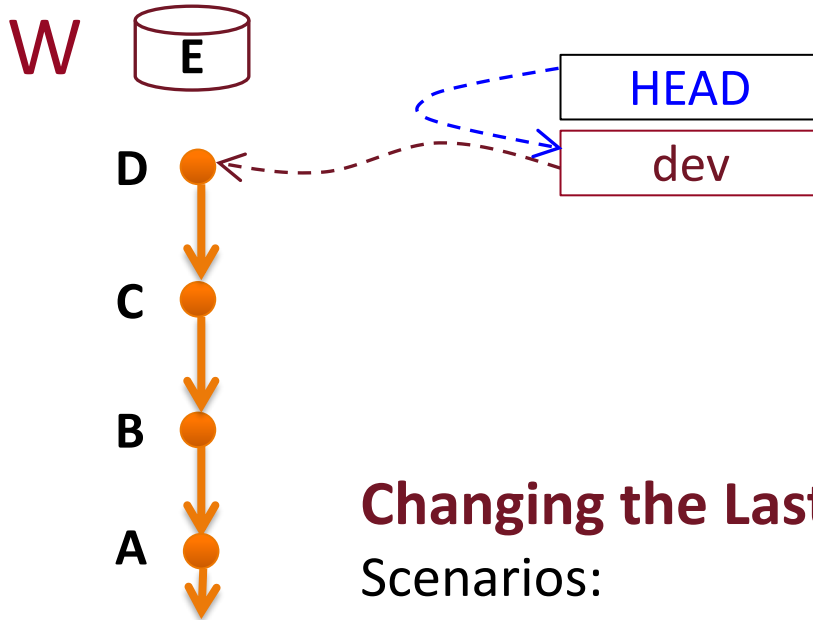
```
git branch my-new-branch
git checkout my-new-branch
# Make the changes in code.
git add <changed files>
git commit -m "My commit message"
git push origin my-new-branch
# Open the merge request link (can also open it from GitHub)
# Choose the branch to merge to; submit merge request.
```

```
# 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
```

Correcting Commits

Amend

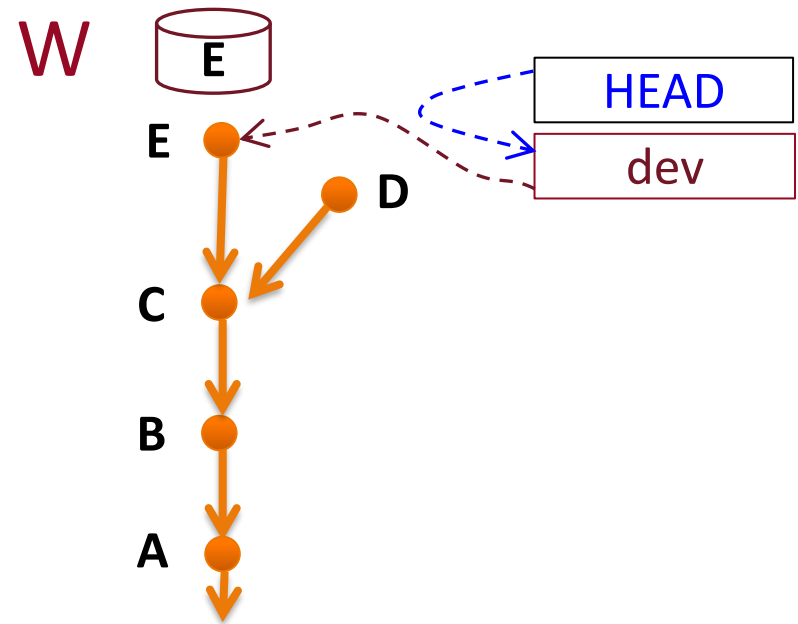
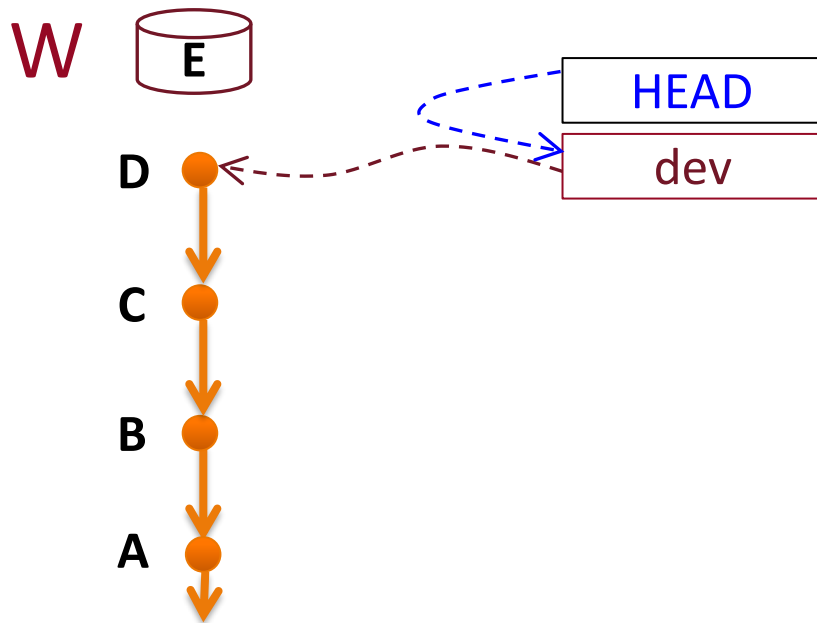


Changing the Last Commit

Scenarios:

- You committed D, then made changes to W (i.e., E)
 - You realize that you committed D too early
 - For example, you forget to add one of the files the first time around
 - Want to include the changes to E
- – Or, you want to change the git commit message of D

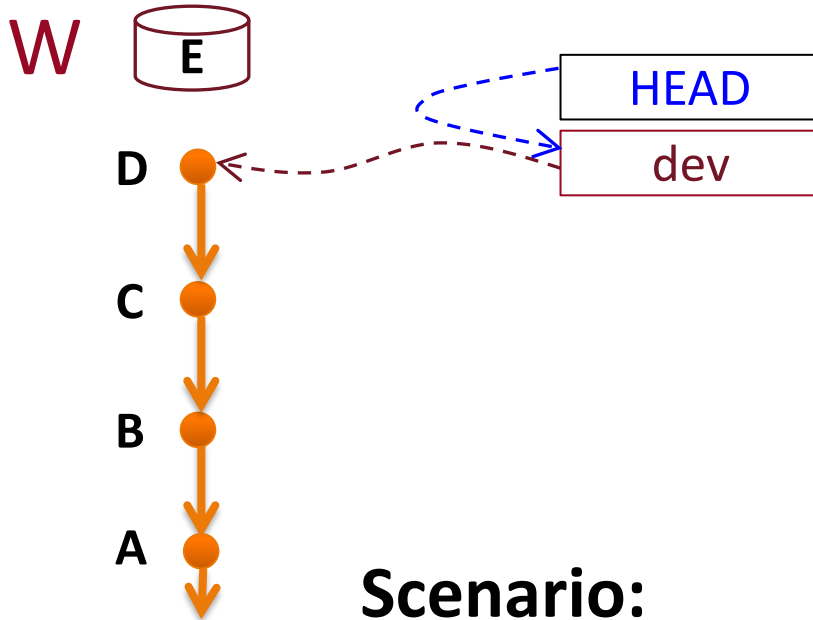
Amend



dev> git commit -m "..." **--amend**

- The commit is amended
 - Includes old changes (C-D) and new ones (D-E)
 - Can change the commit message, and date
 - Sometimes used only to update the commit message
- – Old commit is not deleted; new one is created
 - With git you (almost) never loose commits !

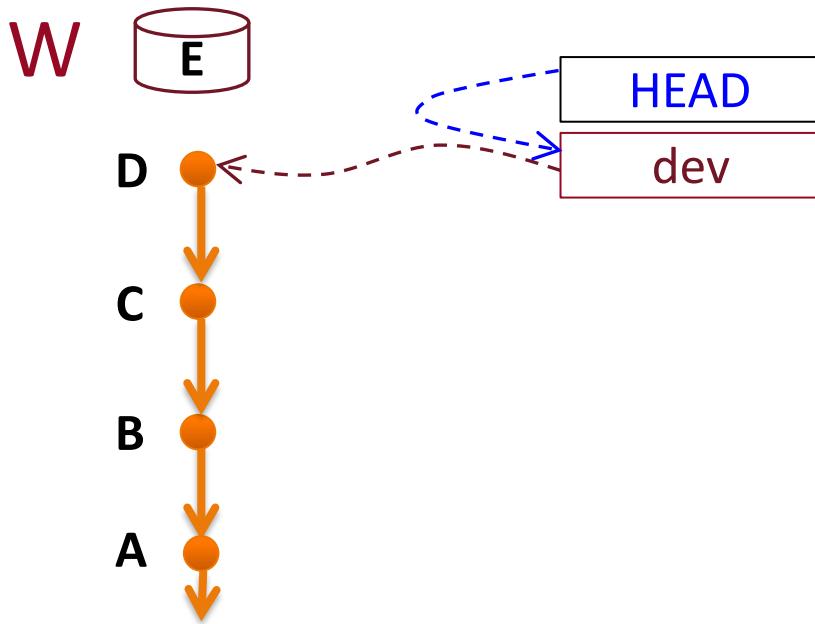
Reset: Un-commit



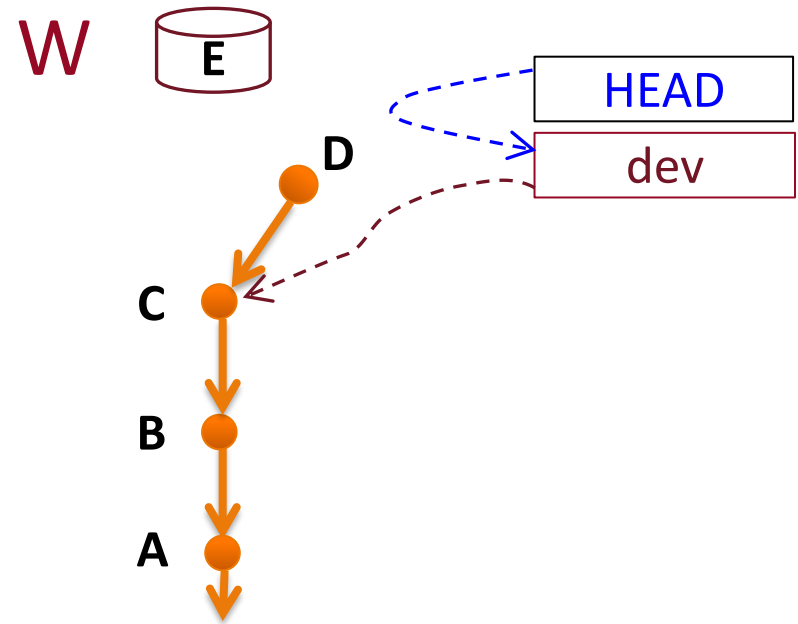
Scenario:

- You want to undo the commit of D, so you can continue working on it
- – Or, you want to change the parent of E (you want to point HEAD somewhere else)

Reset: Un-commit

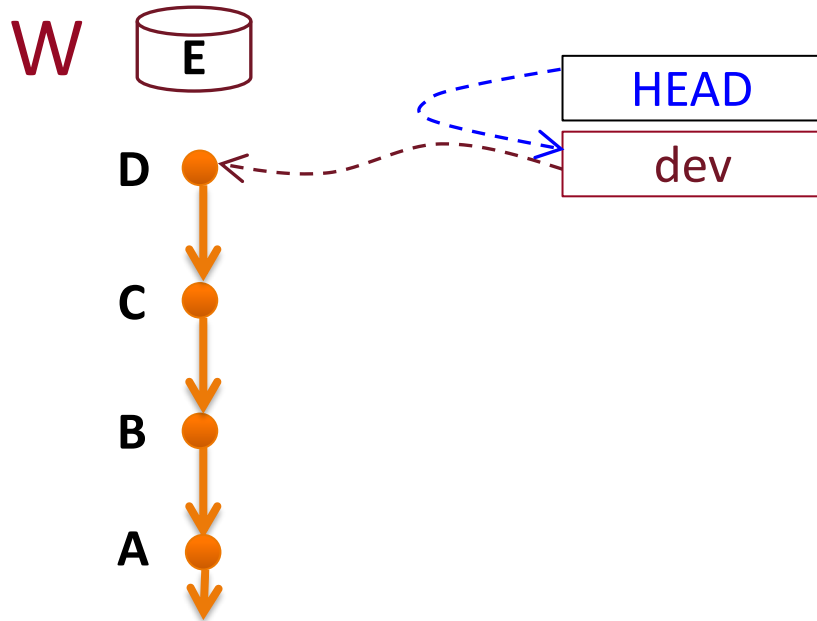


`dev> git reset C`



- The HEAD moves, Workdir is not changed !
 - If you commit now, the parent will be C
 - That would be like “amend”
- Old commit D is still there

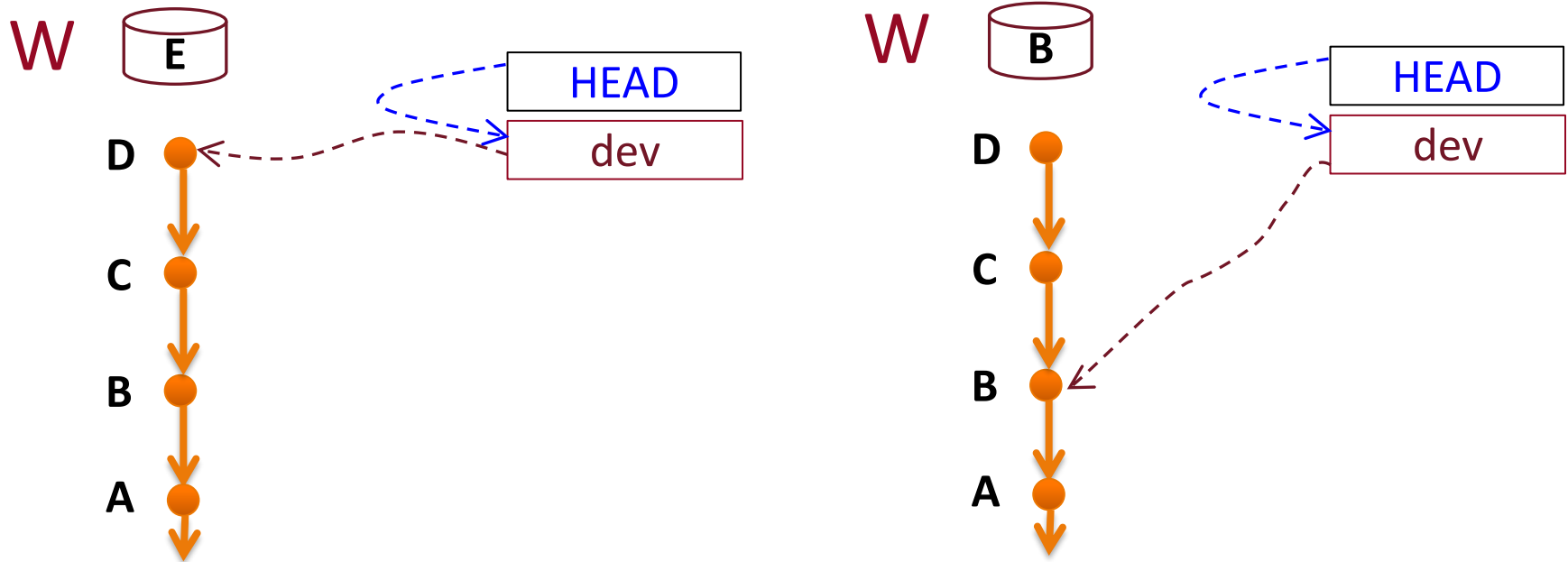
Throwing Away Workdir Changes



Scenario:

- You want to throw away commits C and D, and the changes you have made in Workdir
 - An experiment that went badly
 - Or you screwed up the stuff in Workdir
- Want to get back to commit B (or some other place)

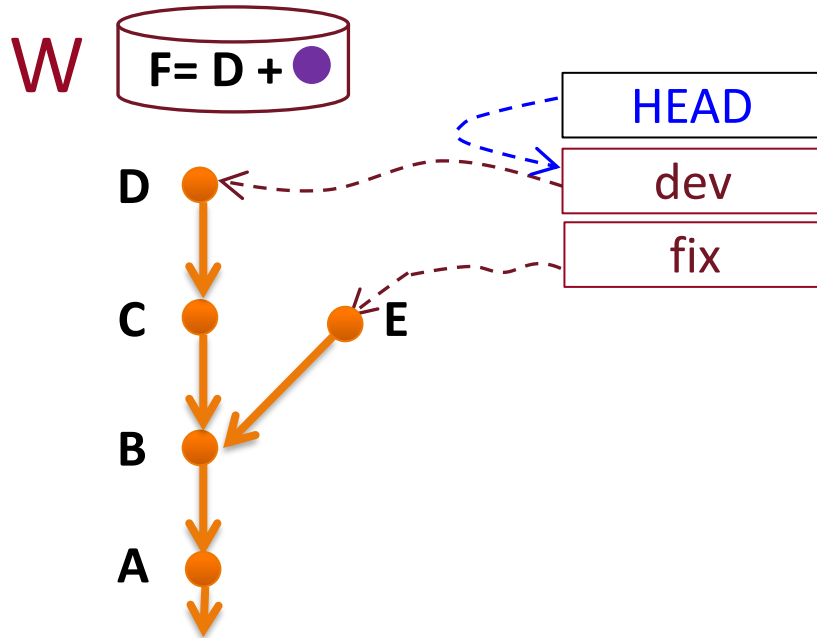
Throwing Away Workdir Changes



`dev> git reset --hard B`

- The HEAD moves, Workdir is changed !
 - The state is like immediately after creating commit B
 - The commits C and D are not lost; just not referenced
 - “`git reset -hard HEAD`” throws away all W changes
 - And loses them forever (they were not in the repo)

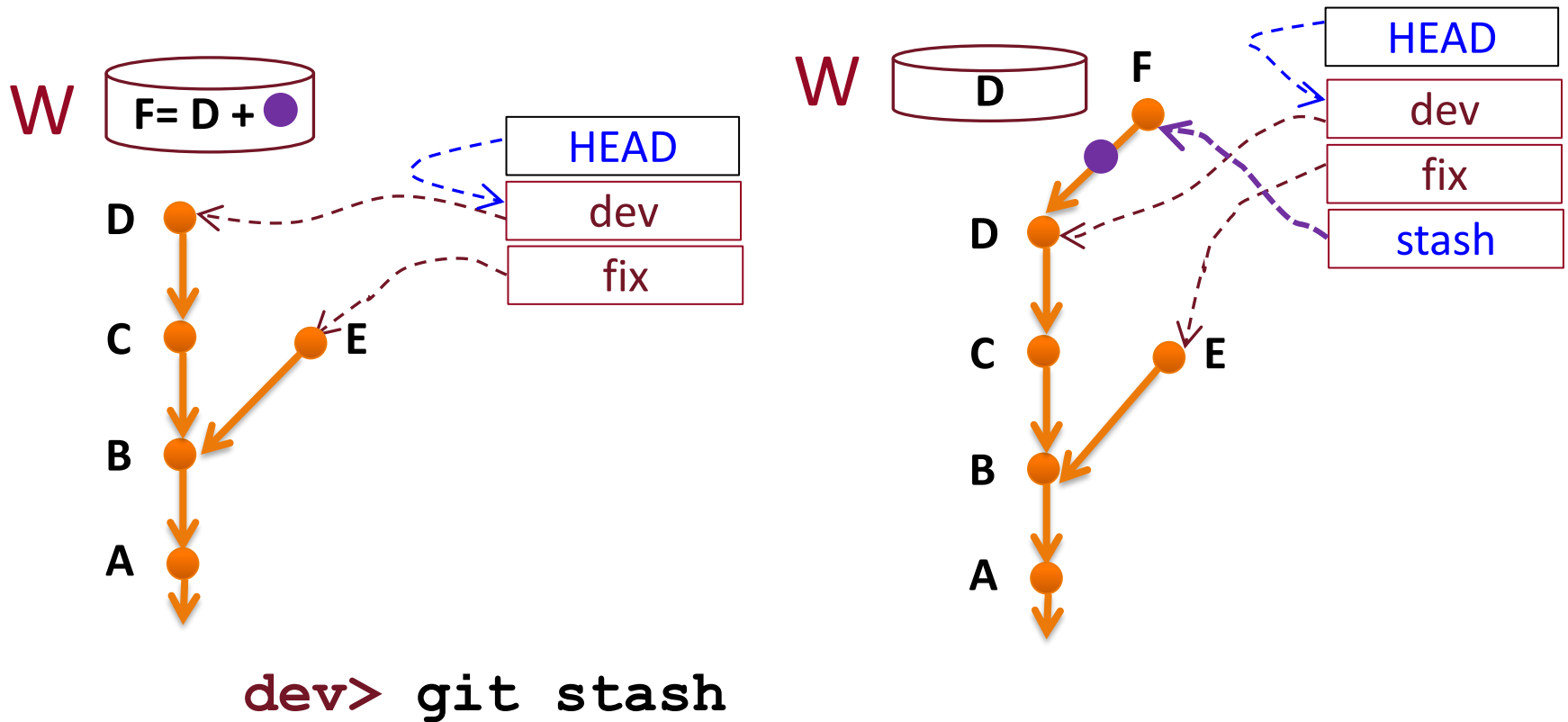
Stash: Temporary Saving Workdir



Scenario:

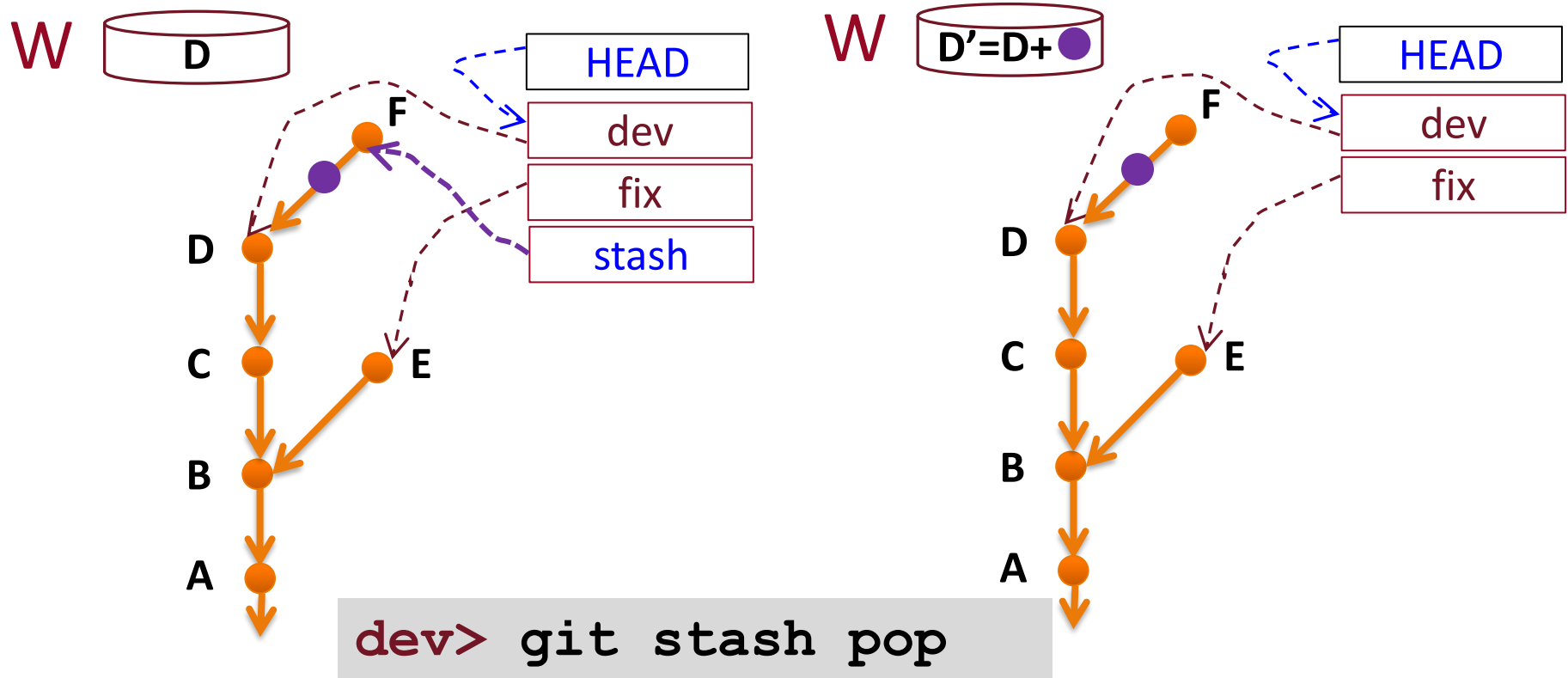
- You are in a middle of change (●) in Workdir (F)
- Need to fix a bug in another branch
 - Want to save the work F
 - Remember: only stuff in repo is safe against loss

Stash: Temporary Saving Workdir



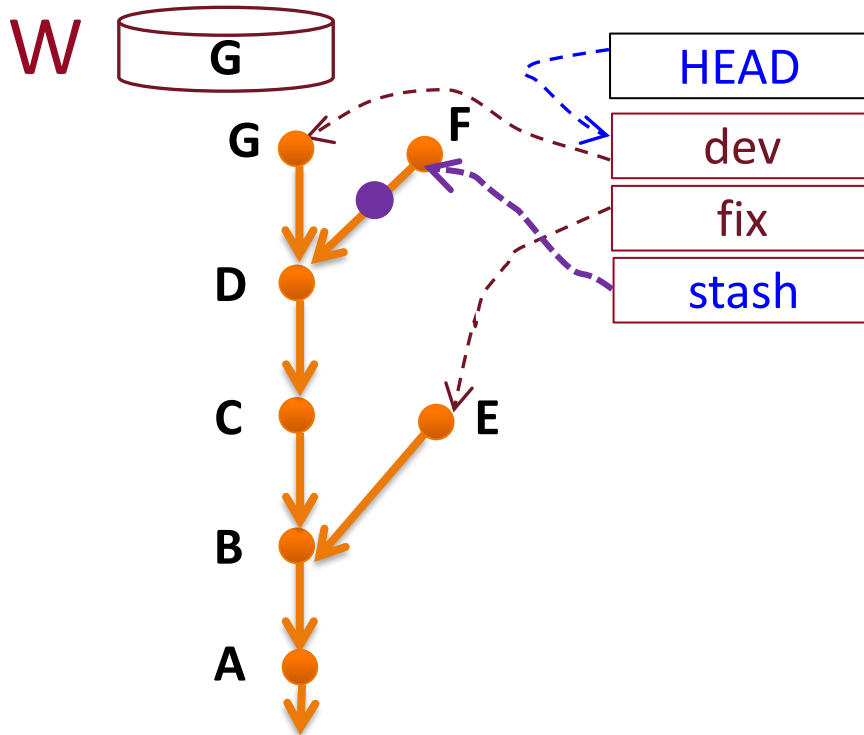
- Saves the W changes from HEAD in the repo
 - HEAD remains the same
 - Then removes changes (●) from Workdir
 - You can now switch branches, etc. without losing your Workdir changes

Restoring Stashed Changes



- **Applies changes from stash to Workdir**
 - Applies changes D-F to D and obtains D' in the Workdir
 - Changes only in Workdir
 - Can continue working with changes from before stashing
 - Since we used “pop” we “used up” the stash

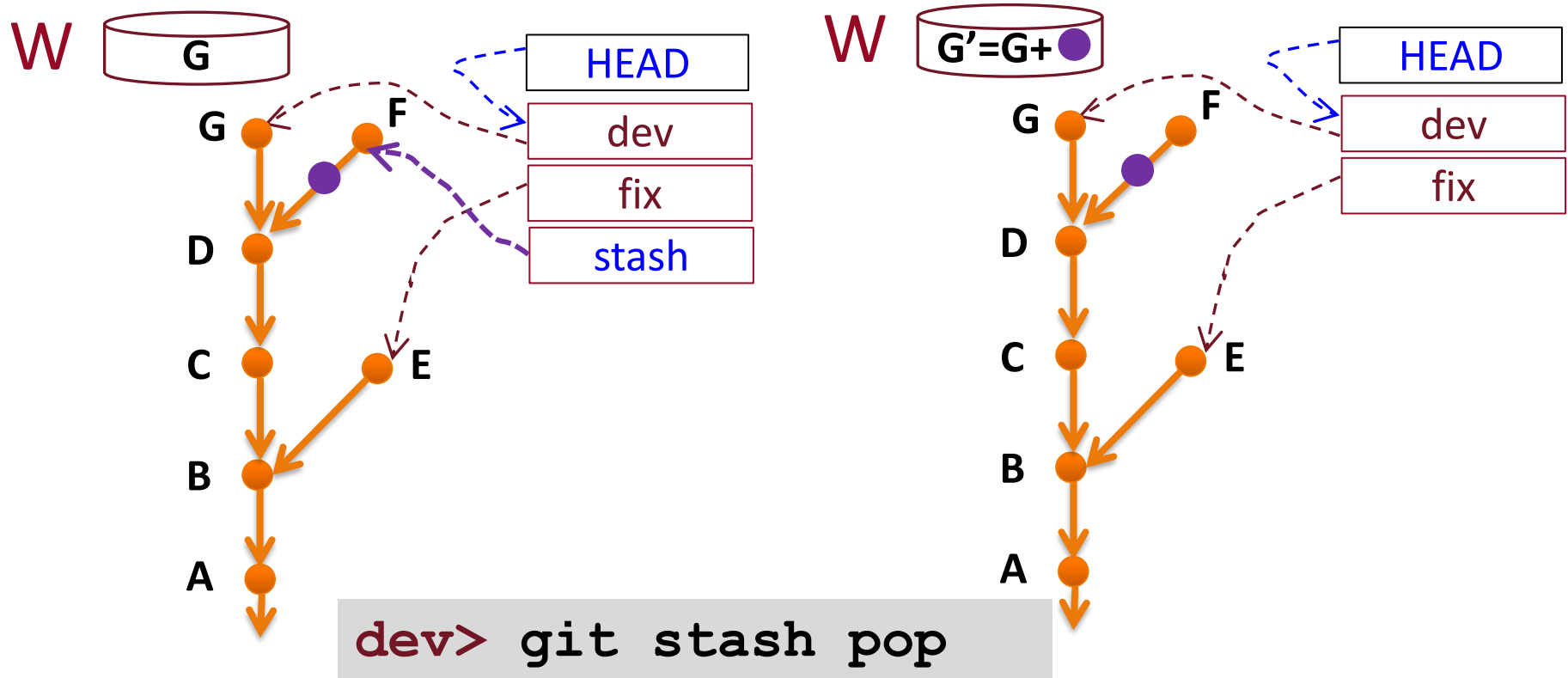
Restoring Stashed Changes



Scenario:

- Has stashed some changes (F) that were on top of D
 - Meanwhile the branch “dev” advanced to G
 - Want to continue working on those changes

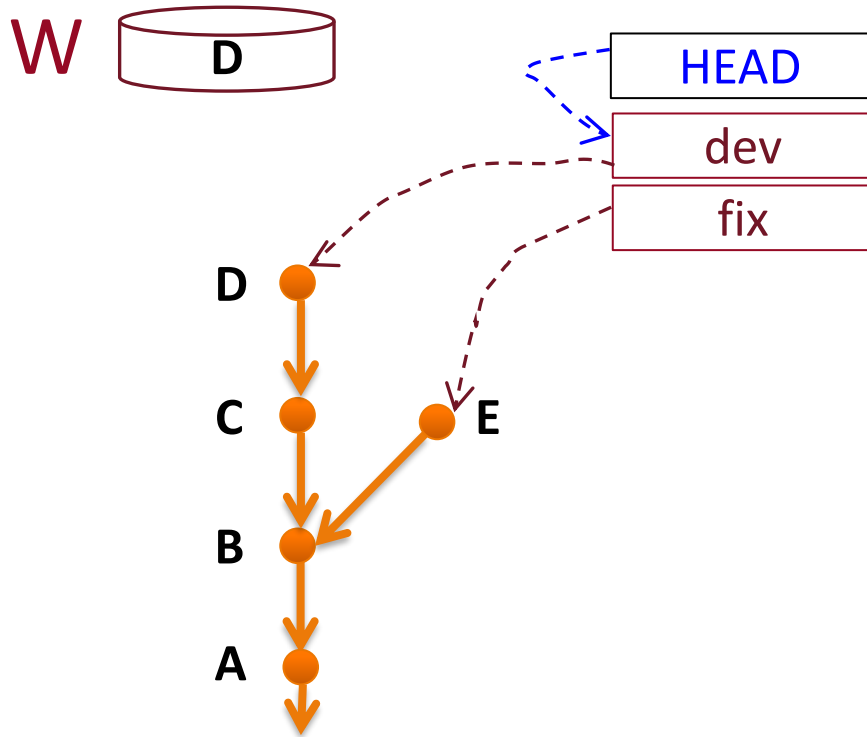
Restoring Stashed Changes



- **Applies changes from stash to Workdir**
 - Applies changes D-F to G and obtains G' in the Workdir
 - Changes only in Workdir
 - Can continue working with changes from before stashing
 - Since we used “pop” we “used up” the stash

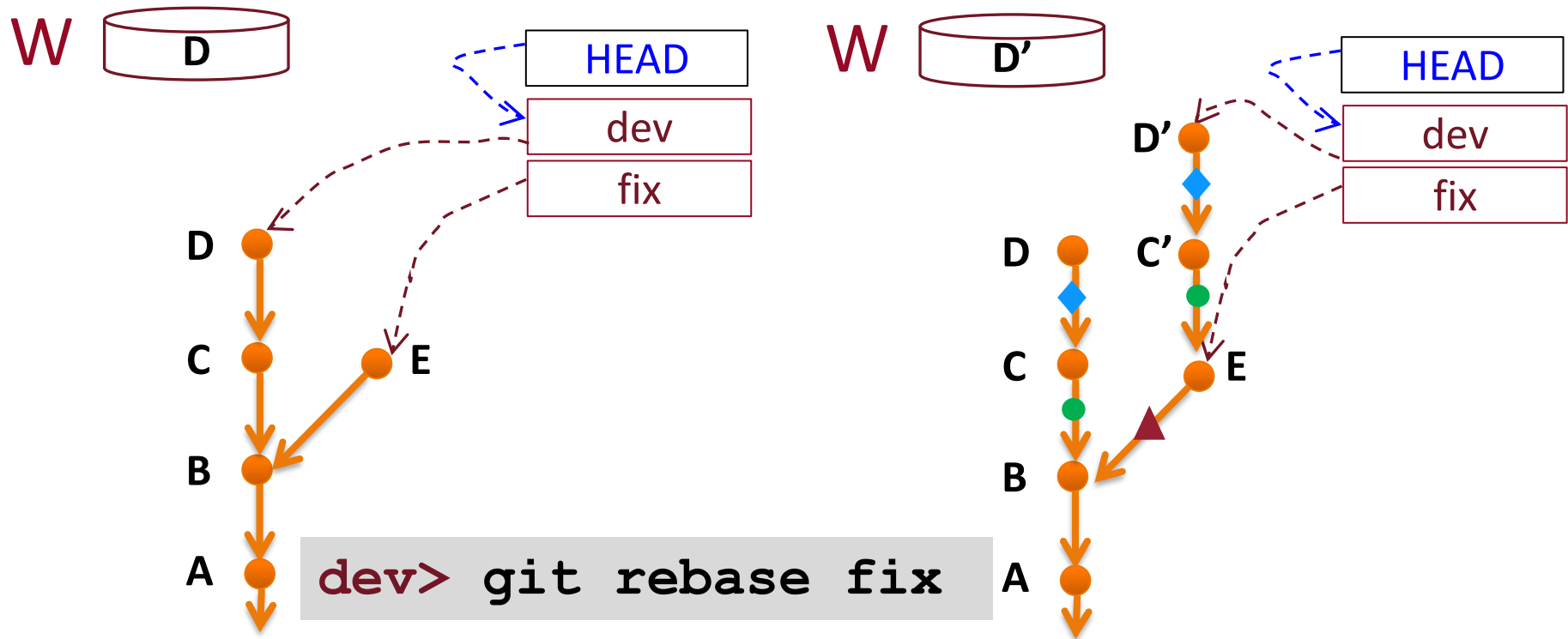
Replaying Commits

Rebase – Copy Commits



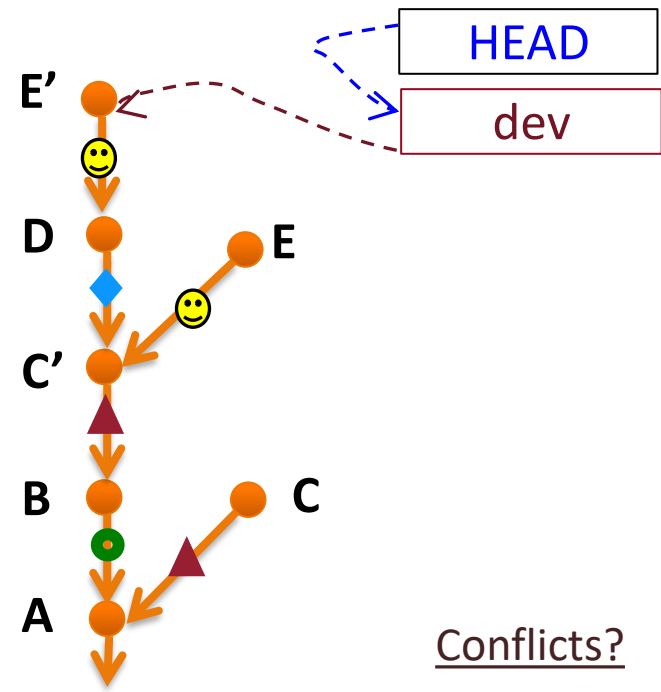
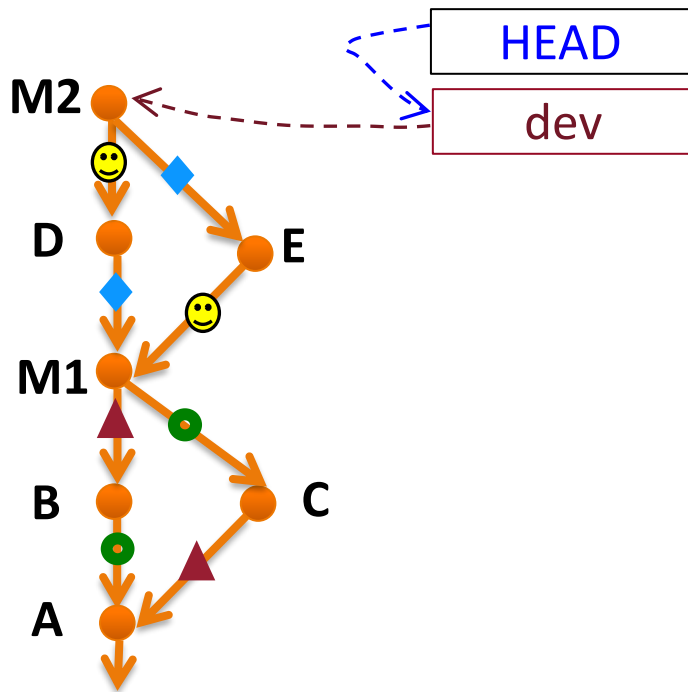
- **Scenario:**
 - Want to combine the changes in “fix” and “dev”
 - Like merging
 - But you want to pretend you did first E then C, then D
 - Want to “replay” changes C and D on top of E

Rebase – Copy Commits



- **Commits in “dev” and not in “E” are replayed on top of “E”**
 - The changes are replayed (B-C then C-D)
 - New commits C' and D' are generated
 - D' has same content with the result of merge D and E
 - Old commits C and D are still there, just not reachable

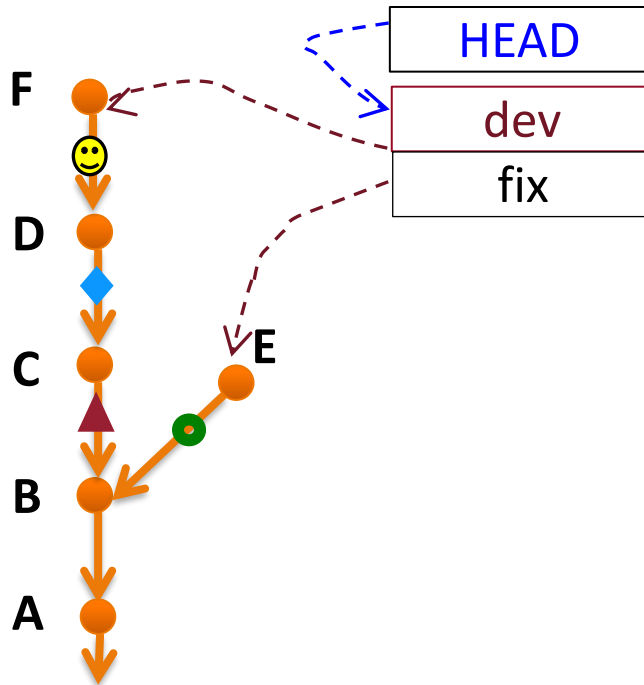
Merge vs. Rebase



Conflicts?

- **Repeated "merge" results in many merge commits**
 - Lots of "diamonds" in the history
 - Each has an ugly message "merge branch 5fee3.."
 - "git log" lists 7 commits

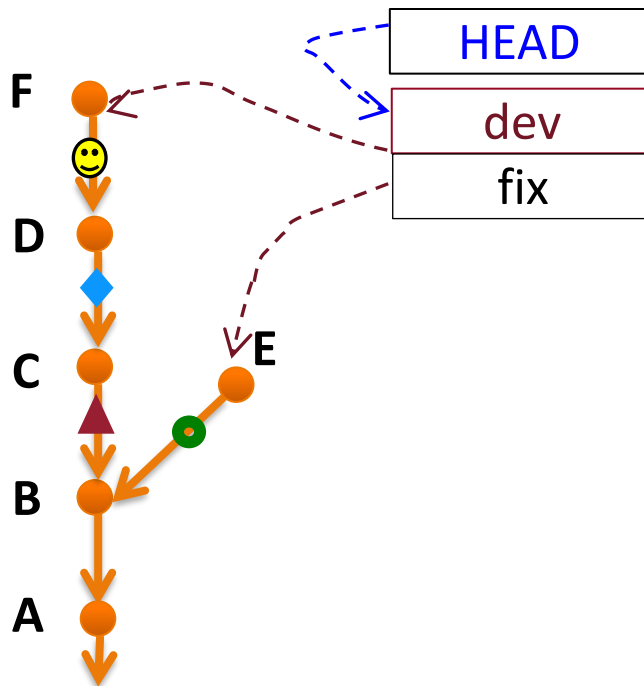
Rebase Onto



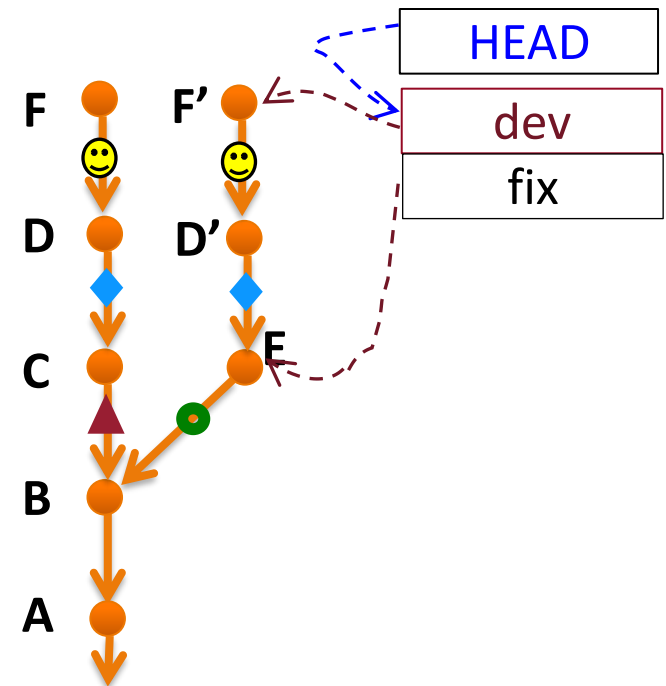
- **Scenario:**

- Want to apply commits E, D and F but not C
 - Ignore B-C

Rebase Onto

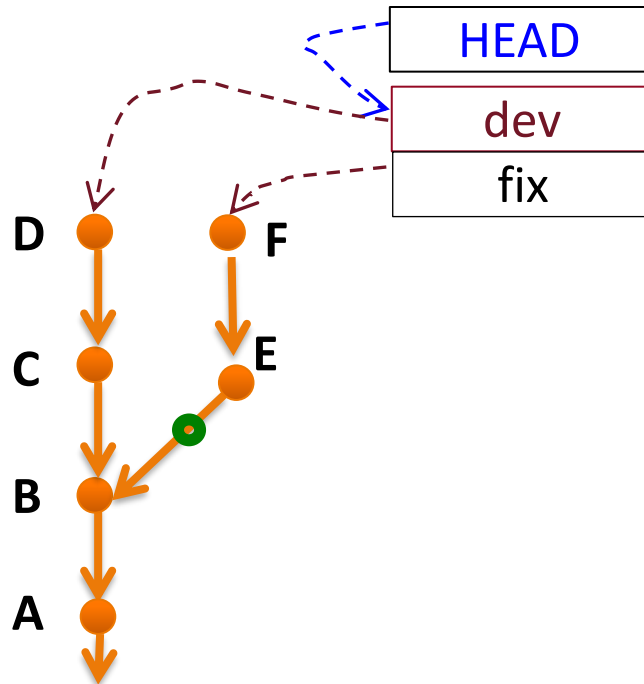


```
dev> git rebase C --onto E
```



- Commits in “dev” and not in “C” are replayed on top of “E”
 - The changes are replayed (C-D then D-F)
 - New commits D' and F' are generated
 - Good for copying changes to another branch

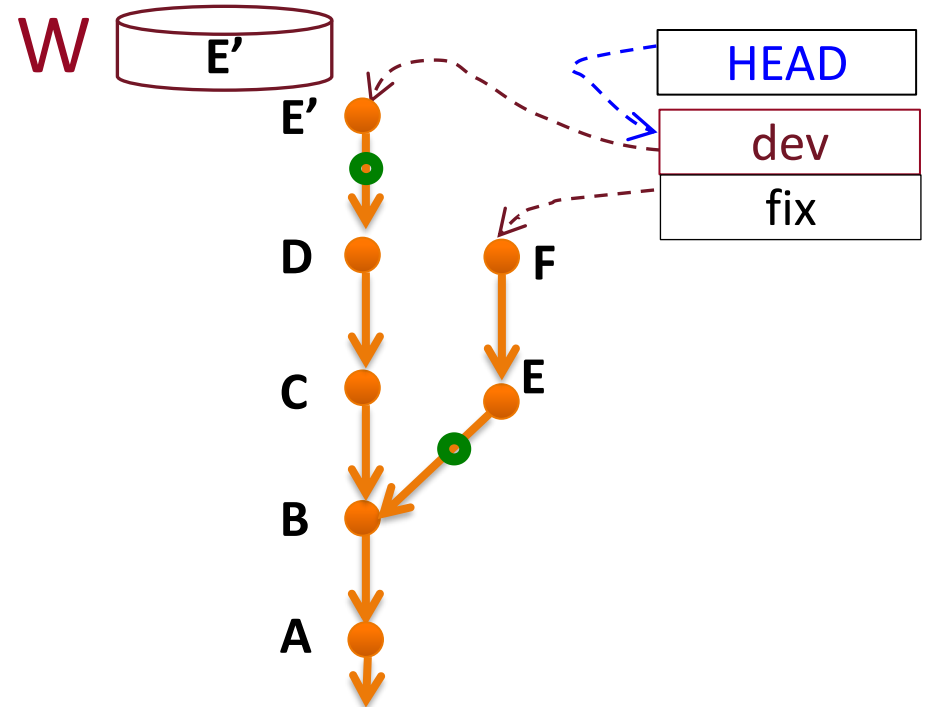
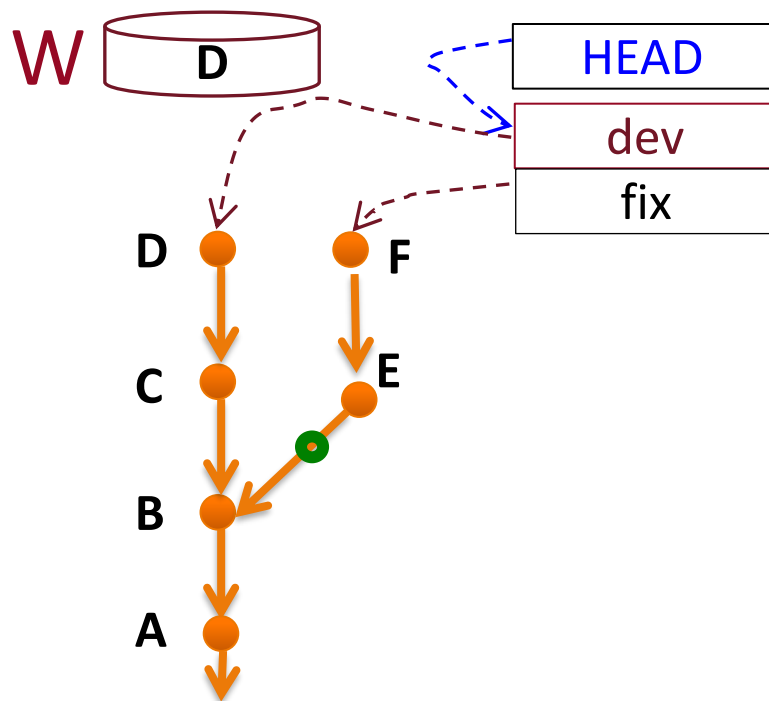
Cherry-Picking



- **Scenario:**

- Want to “cherry-pick” exactly one commit that you want to copy (B-E) on the current branch
 - Maybe E needs to be ported to the “dev” release, without F

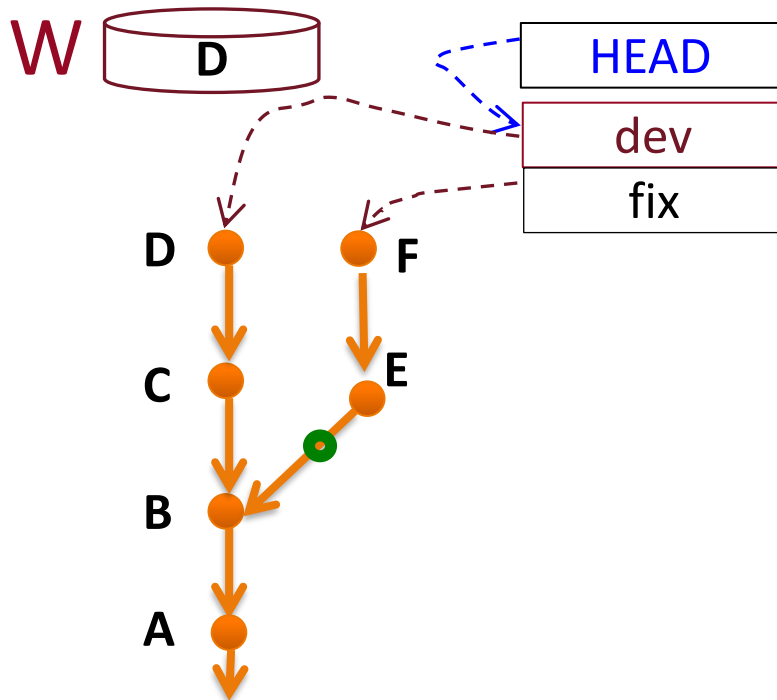
Cherry-Picking



```
dev> git cherry-pick E
```

- **Fine grained copying of commits**
 - The changes in E (B-E) are replayed at HEAD
 - Change D-E' are the same as B-E
 - Good for fine-grained moving changes to another branch

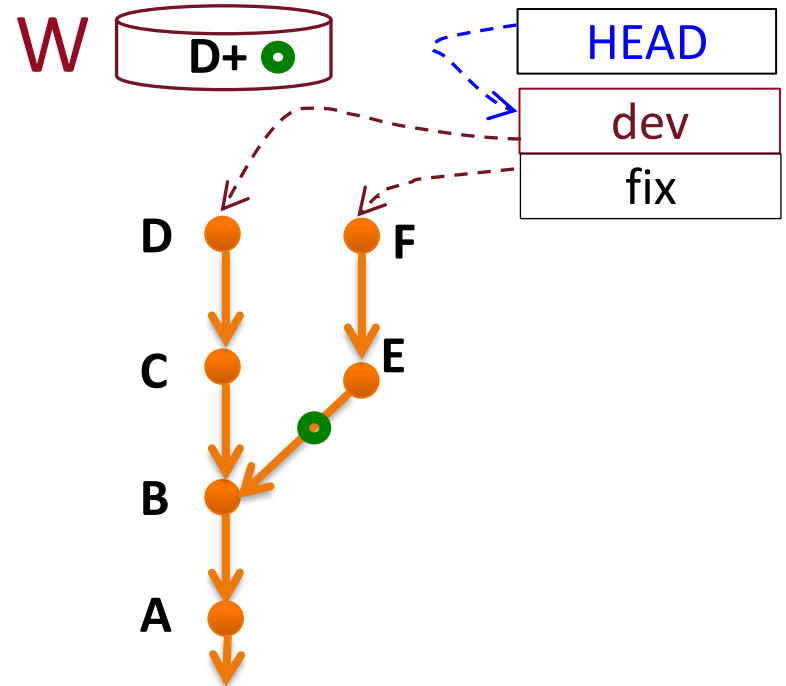
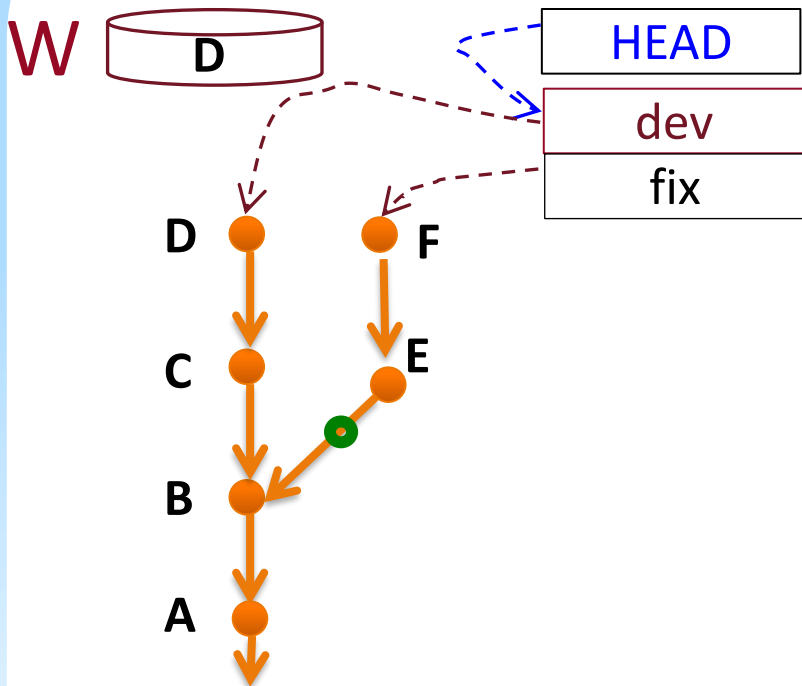
Cherry-Picking -2



- **Scenario:**

- Want to “cherry-pick” exactly the commit that you want to copy (B-E) on the working dir, without committing
 - E.g., want to edit it further, before re-committing
 - E.g., want to do the same with another commit, to combine

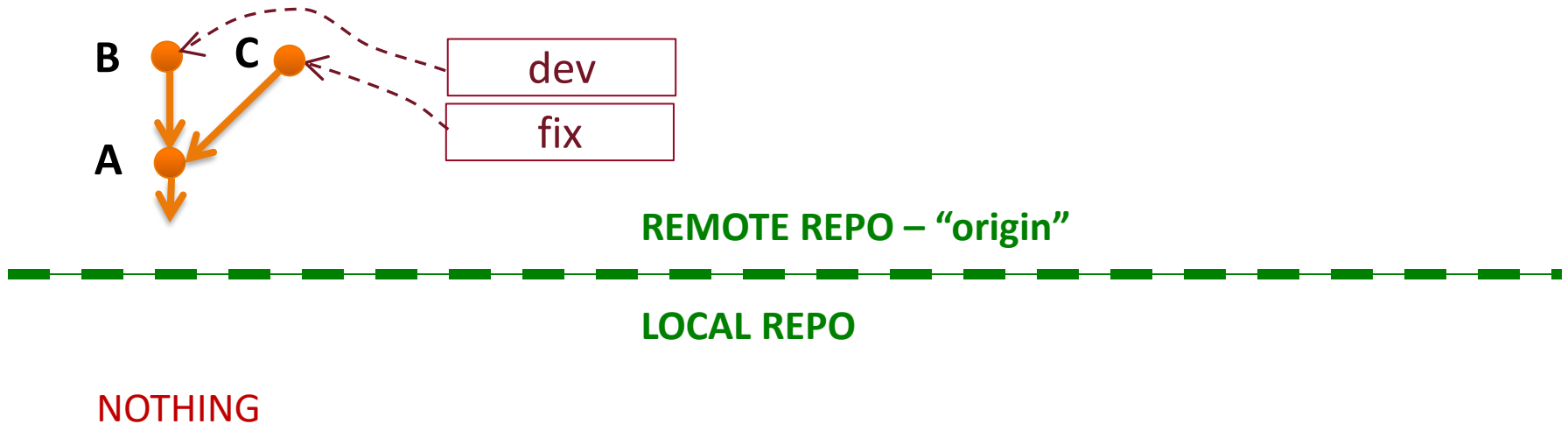
Cherry-Picking -2



```
dev> git cherry-pick -n E
```

- **Applying old commits to Workdir (no commit)**
 - The changes in E (B-E) are replayed at HEAD
 - Change D-E' are the same as B-E
 - Good for fine-grained moving changes to another branch

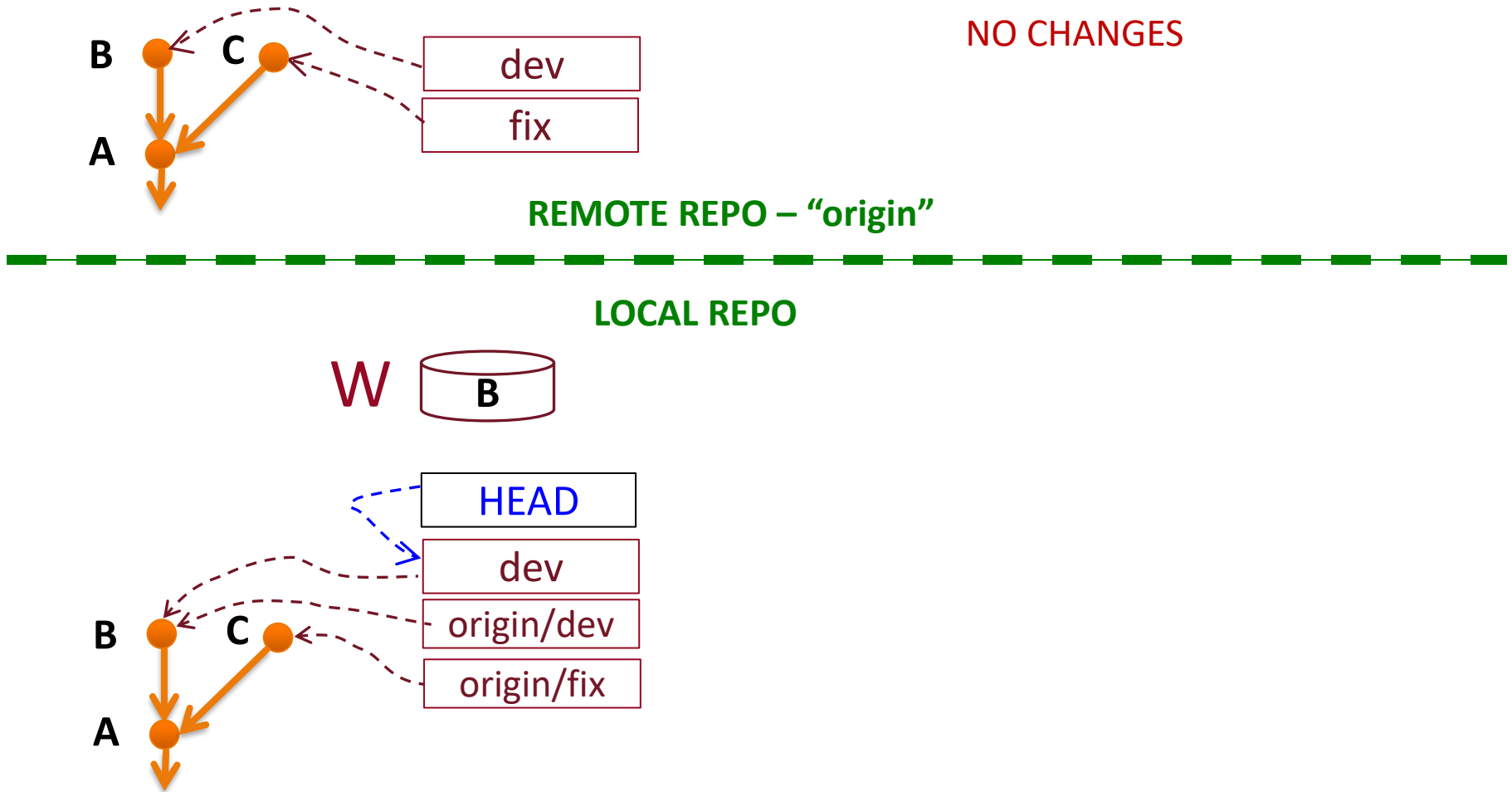
Cloning a Remote Repo



- **Scenario:**

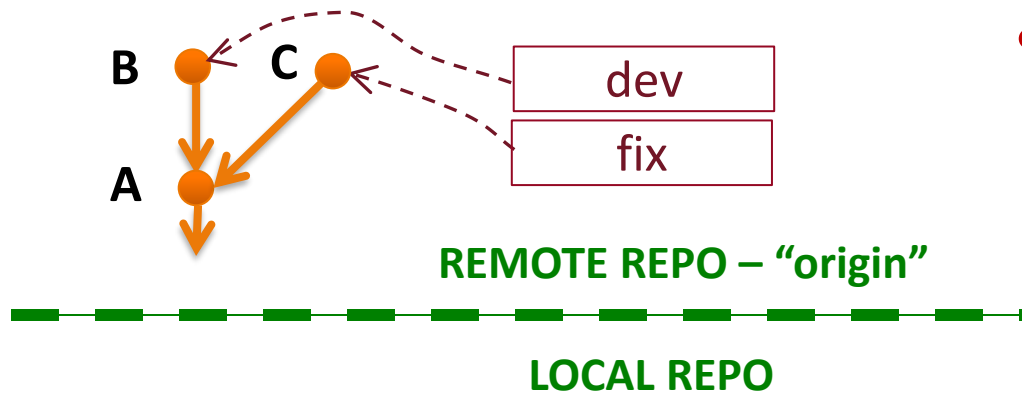
- Want to make a local copy of a remote repo
- So you can checkout/commit
- In git you must have a local copy

Cloning a Remote Repo



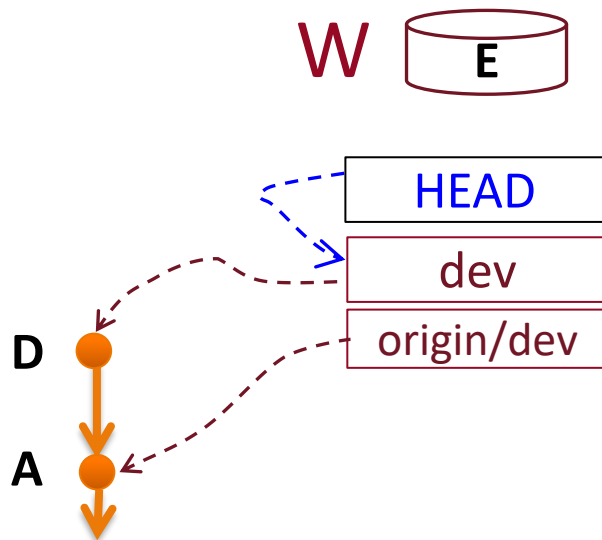
```
dir> git clone http://... -b dev
```

Fetching Updates from a Remote

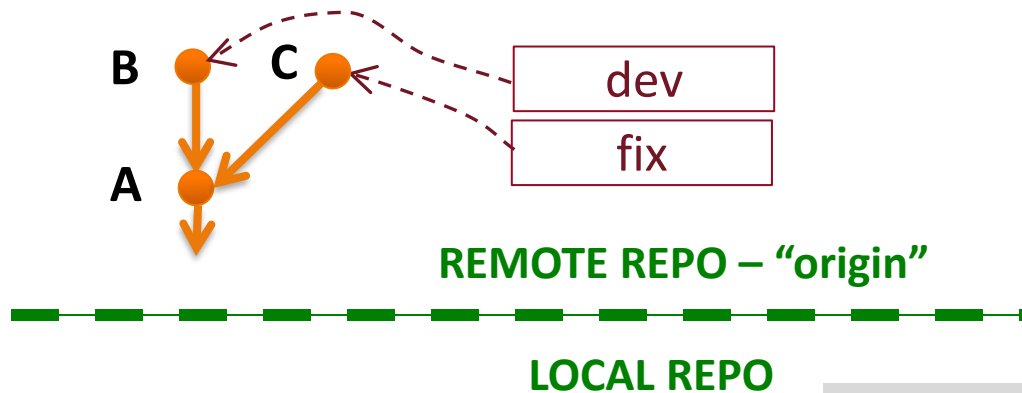


- **Scenario:**

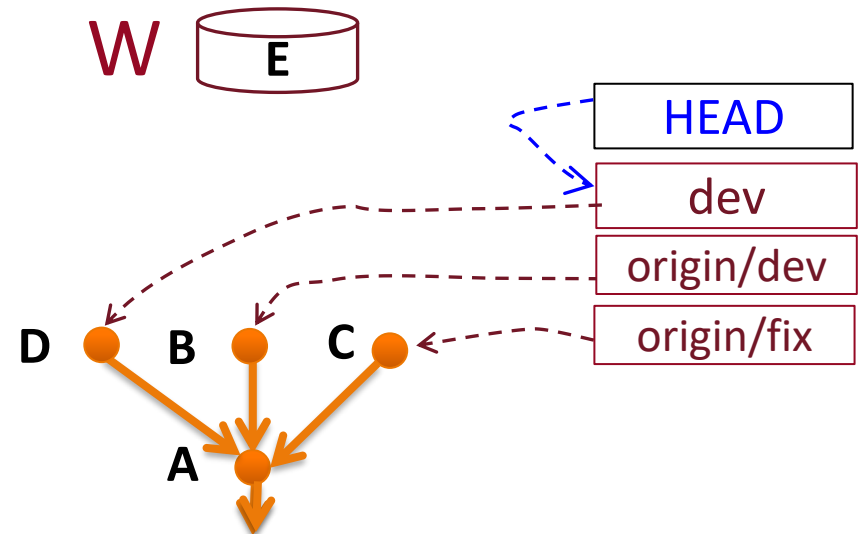
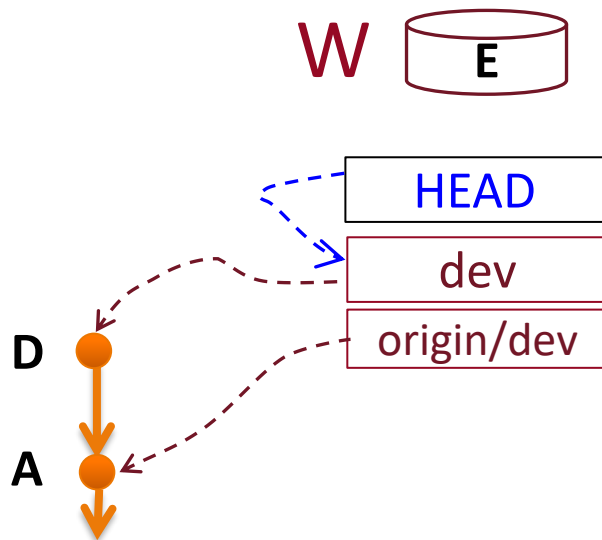
- You cloned remote repo when “dev” was at A
- Remote repo has new branch (fix)
- Remote repo has new commits (B (dev), C (fix))
- Want to bring those changes locally



Fetching Updates from a Remote



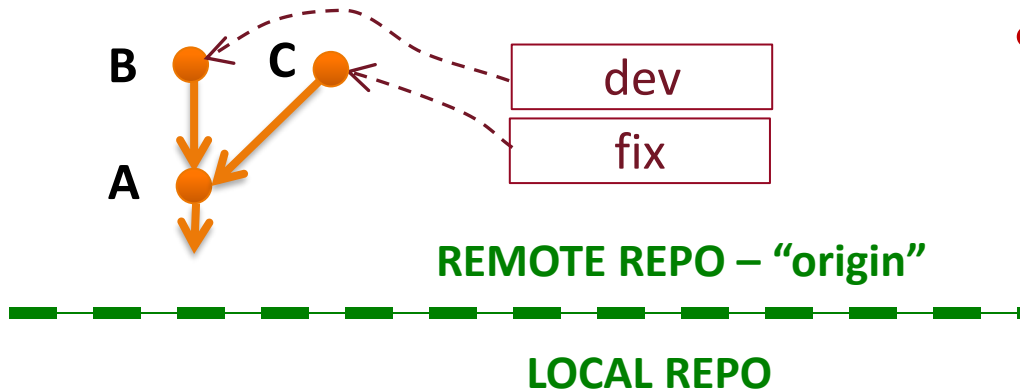
```
dir> git fetch origin
```



Clone vs. Fetch

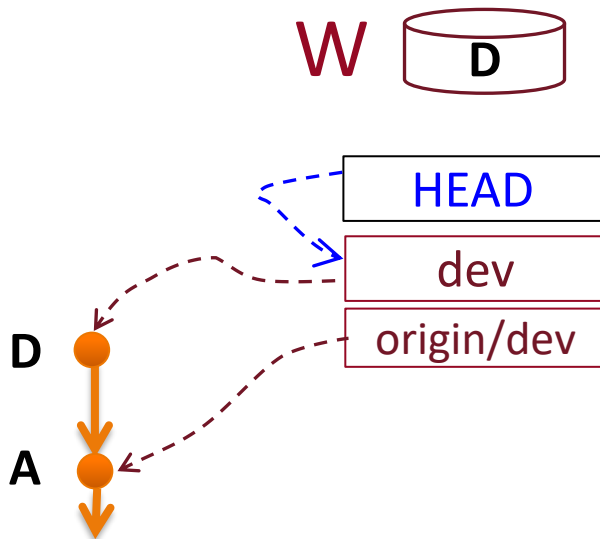
- “git clone http:... -b branch” is like
 1. git init
 2. git remote add origin http://...
 3. git fetch origin
 4. git checkout origin/branch -b branch
- git clone is used only once per repo
- git fetch you run periodically
 - “git fetch” is happening also when you use “git pull”

pull = fetch + merge

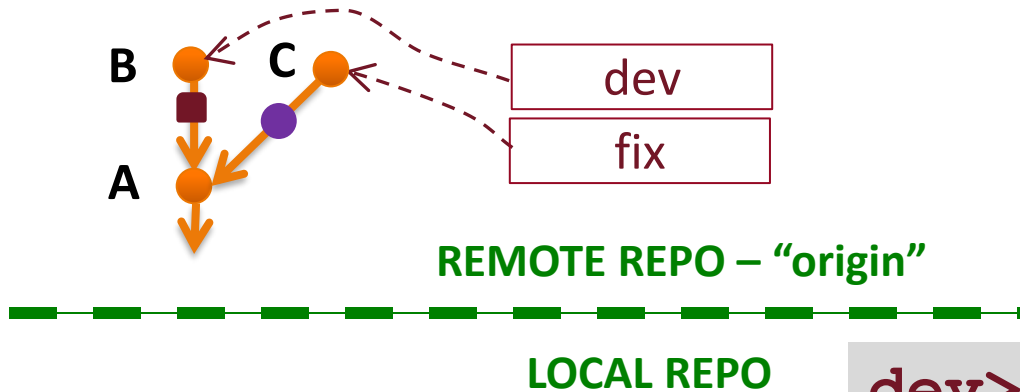


- **Scenario:**

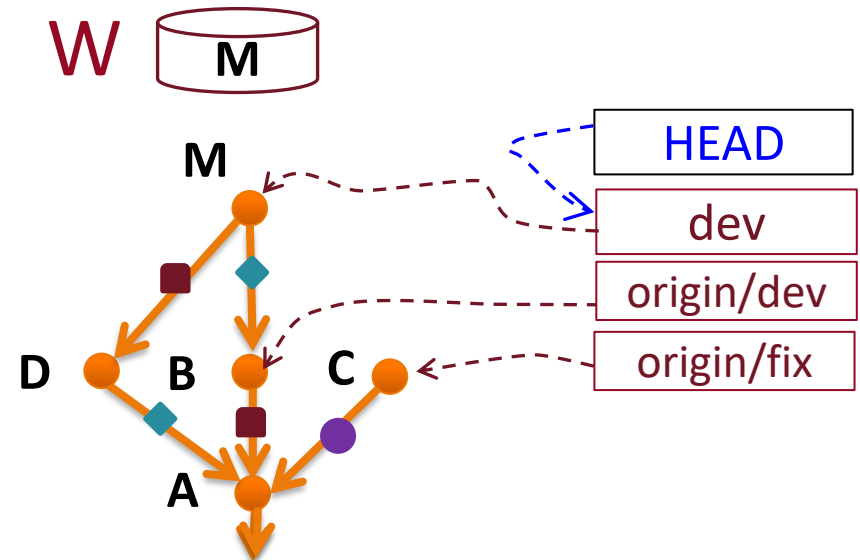
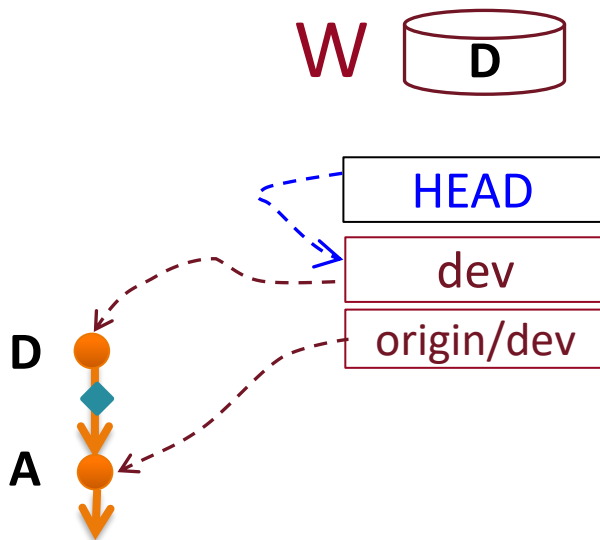
- Want to bring remote changes to local repo (fetch)
- And you want to incorporate changes in current branch (B)
- Like **fetch + merge**



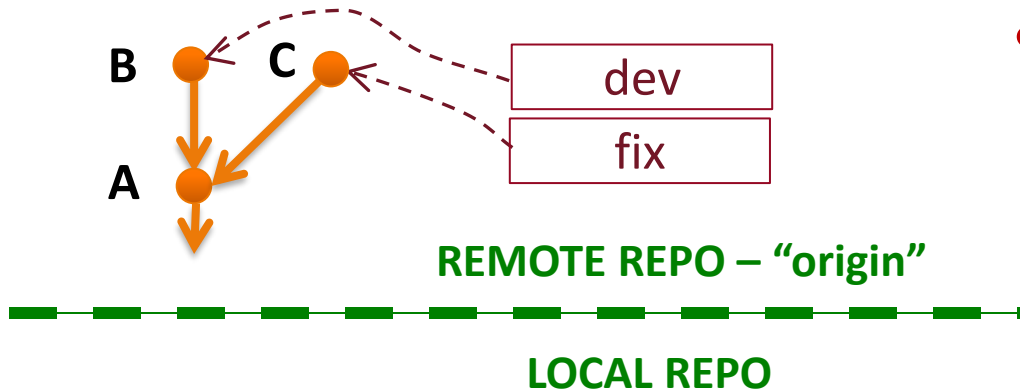
pull = fetch + merge



```
dev> git pull origin dev
```

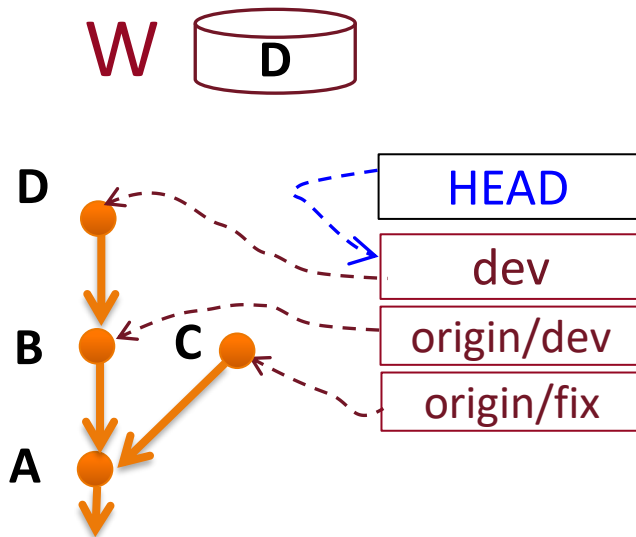


Git push

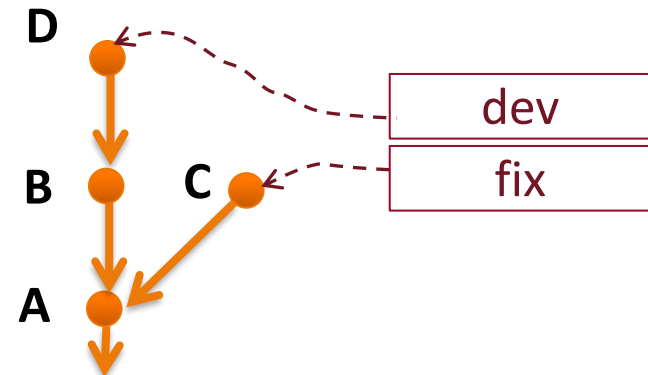
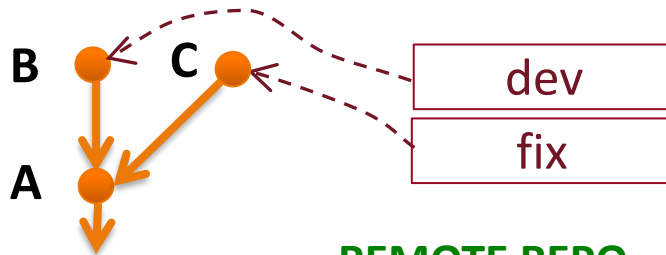


- **Scenario:**

- Have made a local commit (D)
- Want to send it to the remote
 - To share it with others
- Want to advance the remote “dev” branch



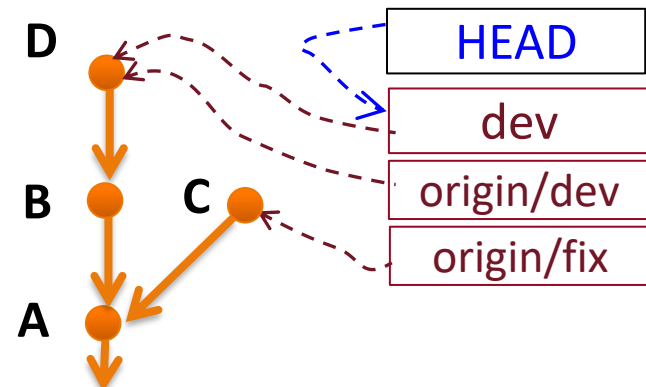
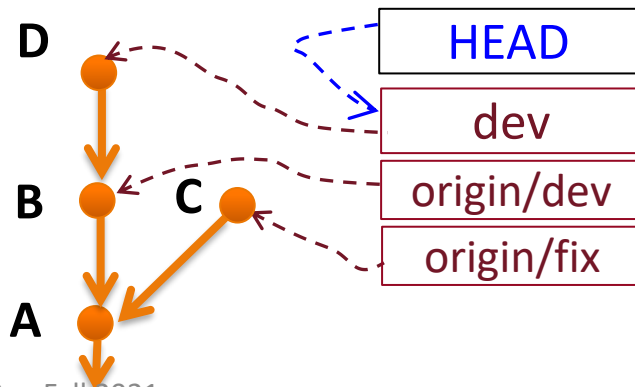
Git push



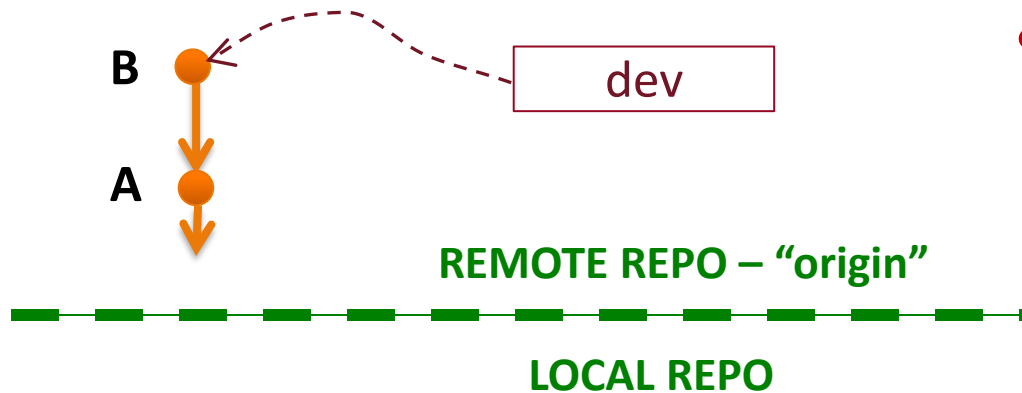
REMOTE REPO - "origin"

LOCAL REPO

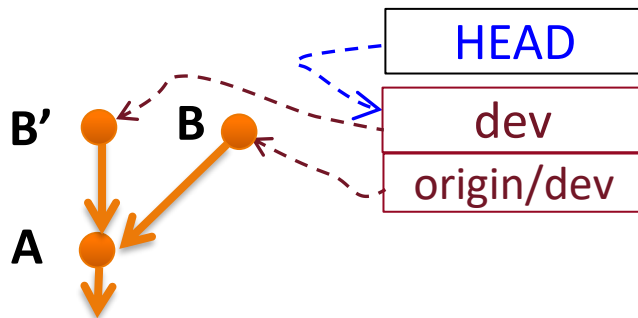
```
dir> git push origin dev
```



Push Errors



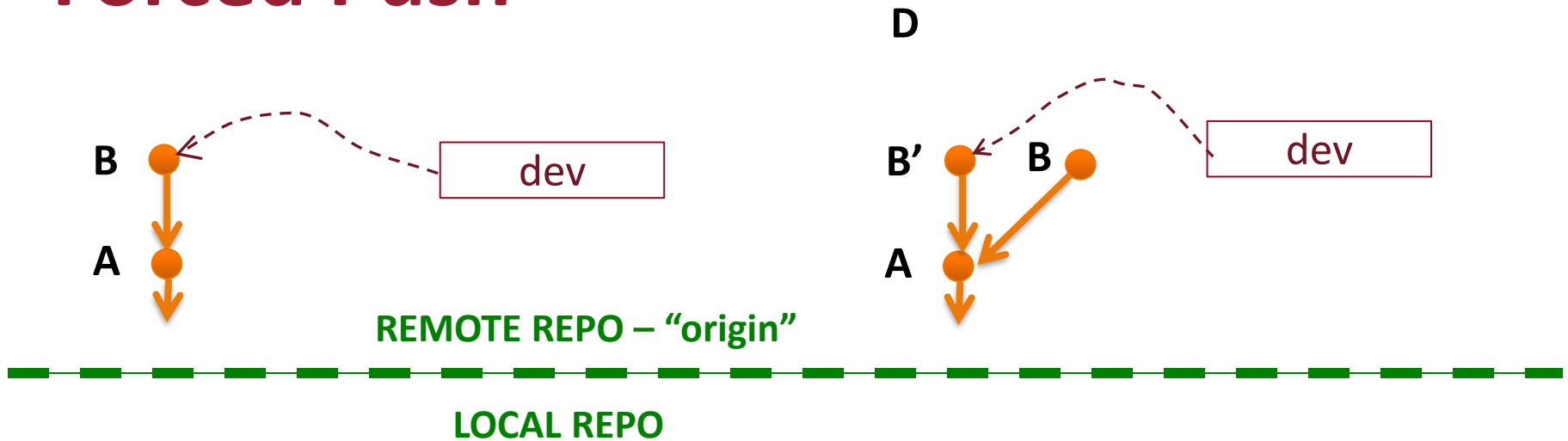
W **B'**



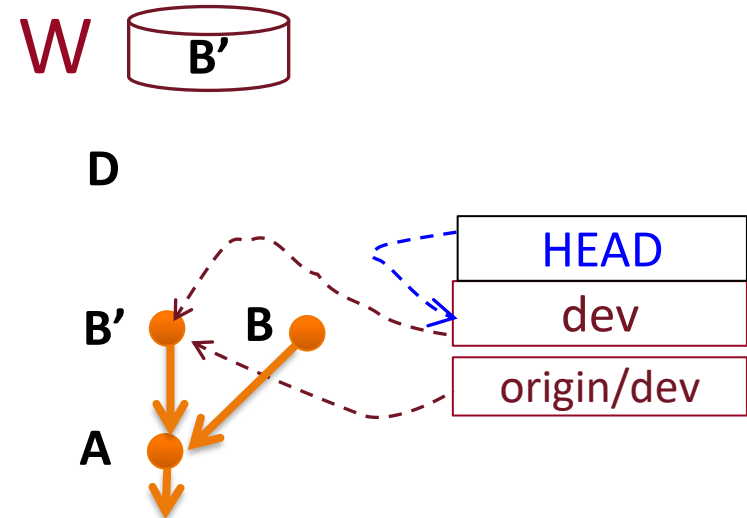
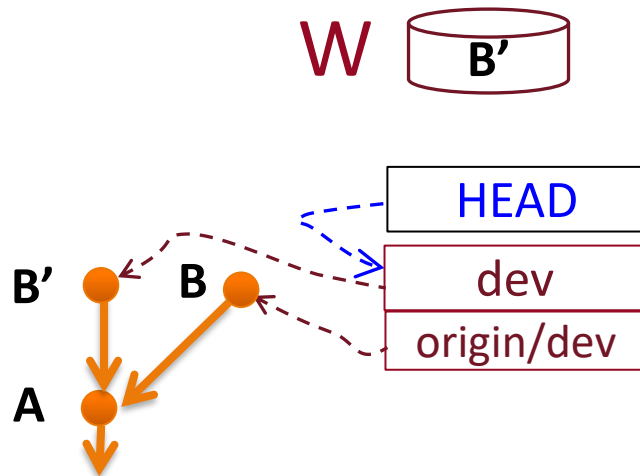
- **Scenario:**

- Have done some commit rewriting in local
 - e.g., Amend
- Push fails because the “dev” branch in origin can only move to descendants
 - Aka fast-forward
- How to enforce the push?

Forced Push



```
dir> git push origin -f
```



GitHub Pull Requests - revisited

- 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 -revisited

```
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
```