git

daniel “paradigm” thau

the ohio state university
open source club
2010-11-12

what is git

what is git

git is a distributed version control system

what is git

git is a distributed version control system

what?

distributed version control systems

languages

you can version control...

languages

you can version control...

- c++

languages

you can version control...

- c++
- python

languages

you can version control...

- c++
- python
- english

languages

you can version control...

- c++
- python
- english
- pretty much anything

languages

you can version control...

- c++
- python
- english ⟵ everyone here knows this one
- pretty much anything

languages

you can version control...

- c++
- python
- english
- pretty much anything

will be used for examples

everyone here knows this one

task 1

our task is to compose an essay

attempt 1: saving

write

attempt 1: saving

write $\longrightarrow$ save

attempt 1: saving

write $\longrightarrow$ save $\longrightarrow$ write

attempt 1: saving

write $\longrightarrow$ save $\longrightarrow$ write $\longrightarrow$ save

attempt 1: saving

write $\longrightarrow$ save $\longrightarrow$ write $\longrightarrow$ save

active copy

penguins are so cute!
but they eat icky fishies

attempt 1: saving

write $\longrightarrow$ save $\longrightarrow$ write $\longrightarrow$ save

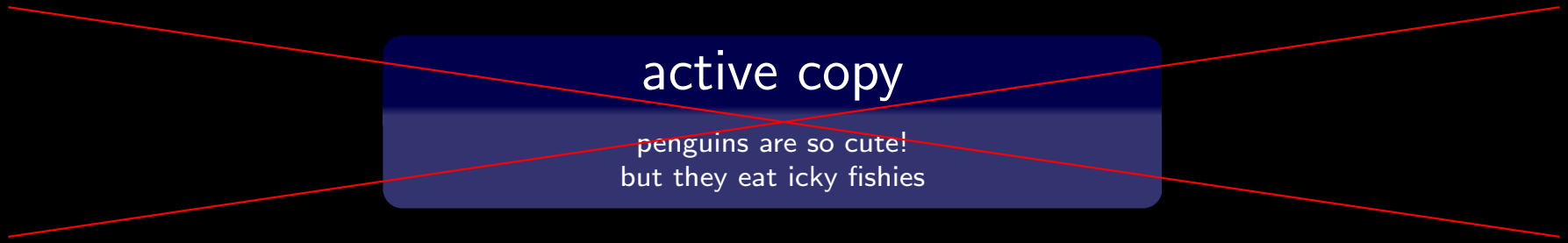
active copy

penguins are so cute!
but they eat icky fishies

computer spontaneously combusts

attempt 1: saving

write $\longrightarrow$ save $\longrightarrow$ write $\longrightarrow$ save



computer spontaneously combusts

attempt 2: backups

active copy

penguins are so cute!
but they eat icky fishies

backup

penguins are so cute!
but they eat icky fishies

attempt 2: backups

active copy

penguins are so cute!
but penguins are so cute!

backup

penguins are so cute!
but they eat icky fishies

we do something dumb in the active copy

attempt 2: backups

active copy

penguins are so cute!
but penguins are so cute!

backup

penguins are so cute!
but they eat icky fishies

we do something dumb in the active copy

back it up

attempt 2: backups

active copy	backup
penguins are so cute! but penguins are so cute!	penguins are so cute! but penguins are so cute!

we do something dumb in the active copy

back it up

no good copies

attempt 3: iterative backups

active copy

penguins are so cute!
but they eat icky fishies

backup 2010-10-08

penguins are so cute!
but they eat icky

backup 2010-10-07

penguins are so cute!
but they eat

backup 2010-10-06

penguins are so cute!
but they

backup 2010-10-05

penguins are so cute!
but

backup 2010-10-04

penguins are so cute!

backup 2010-10-03

penguins are so

backup 2010-10-02

penguins are

backup 2010-10-01

penguins

attempt 3: iterative backups

backups should have metadata:

attempt 3: iterative backups

backups should have metadata:

- date/time

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

backup + metadata = "commit"

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

backup + metadata = "commit"

save some space:

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

backup + metadata = "commit"

save some space:

- deltas

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

backup + metadata = "commit"

save some space:

- deltas (which git does not use)

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

backup + metadata = "commit"

save some space:

- deltas (which git does not use)
- compression

attempt 3: iterative backups

backups should have metadata:

- date/time
- comment about what changed
- who made the change
  - (this will make sense later)

backup + metadata = "commit"

save some space:

- deltas (which git does not use)
- compression (which git does use)

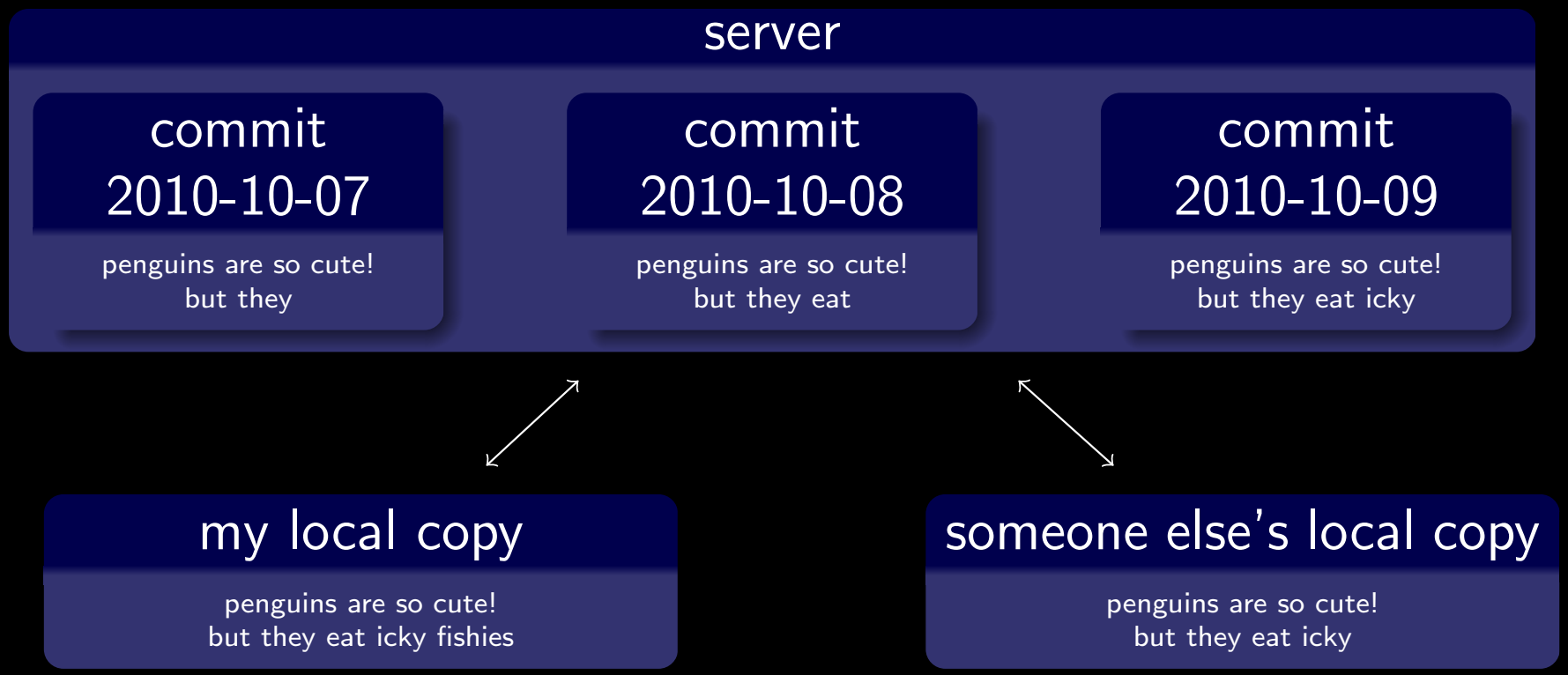
task 2

our task is to compose an essay

task 2

our task is to compose an essay
collaboratively with other people

attempt 4: centralized



attempt 4: centralized

(otherlocal)

(server) (A)

(mylocal)

attempt 4: centralized

(otherlocal)

(server) (A)

↓

(mylocal) (A)

attempt 4: centralized

(otherlocal)

(server) (A)

$$\downarrow$$

(mylocal) (A) $\longrightarrow$ (B)

attempt 4: centralized

(otherlocal)

(server) (A) (B)

$$\begin{array}{ccc} & \downarrow & \uparrow \\ \text{(mylocal)} & \text{(A)} & \longrightarrow \text{(B)} \end{array}$$

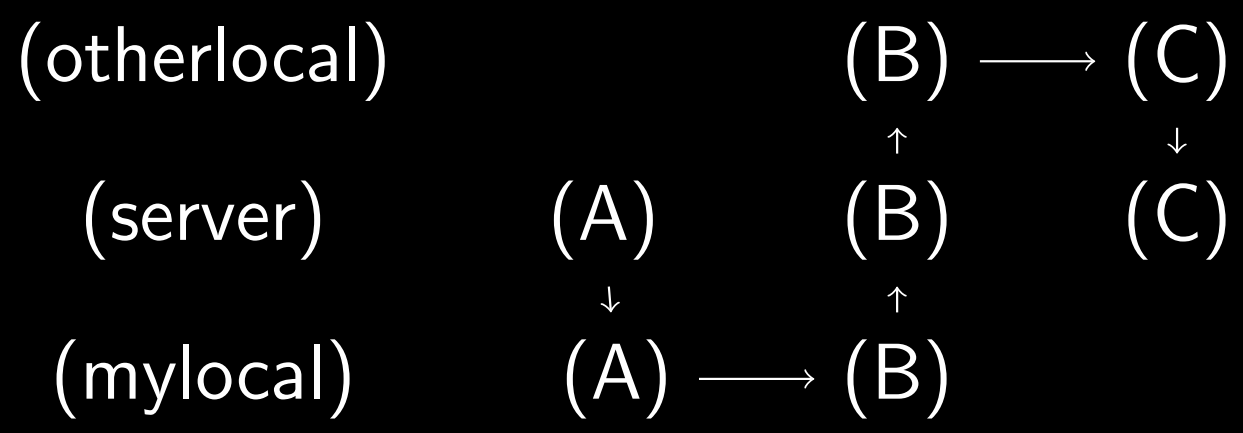
attempt 4: centralized

$$
\begin{array}{lccc}
\text{(otherlocal)} & & & \text{(B)} \\
 & & & \uparrow \\
\text{(server)} & & \text{(A)} & \text{(B)} \\
 & & \downarrow & \uparrow \\
\text{(mylocal)} & & \text{(A)} & \longrightarrow \text{(B)}
\end{array}
$$

attempt 4: centralized

$$
\begin{tikzcd}
(\text{otherlocal}) & & (B) \arrow[r] & (C) \\
(\text{server}) & (A) \arrow[d] & (B) \arrow[u] & \\
(\text{mylocal}) & (A) \arrow[r] & (B) \arrow[u] &
\end{tikzcd}
$$

attempt 4: centralized



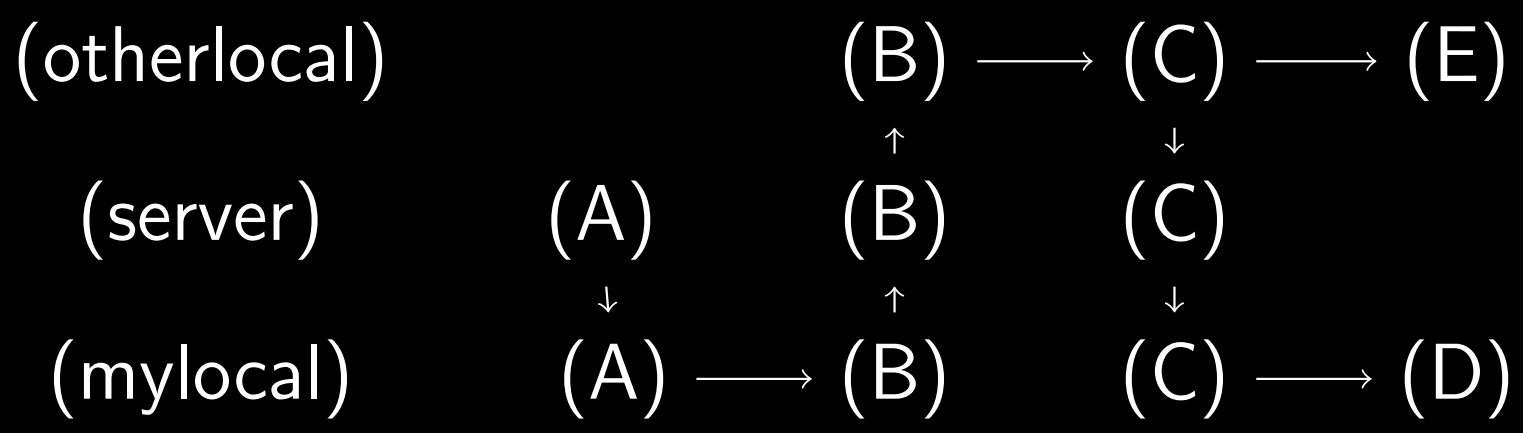
attempt 4: centralized

$$
\begin{tikzcd}
\text{(otherlocal)} & & (B) \arrow[r] & (C) \arrow[d] \\
\text{(server)} & (A) \arrow[d] & (B) \arrow[u] & (C) \arrow[d] \\
\text{(mylocal)} & (A) \arrow[r] & (B) \arrow[u] & (C)
\end{tikzcd}
$$

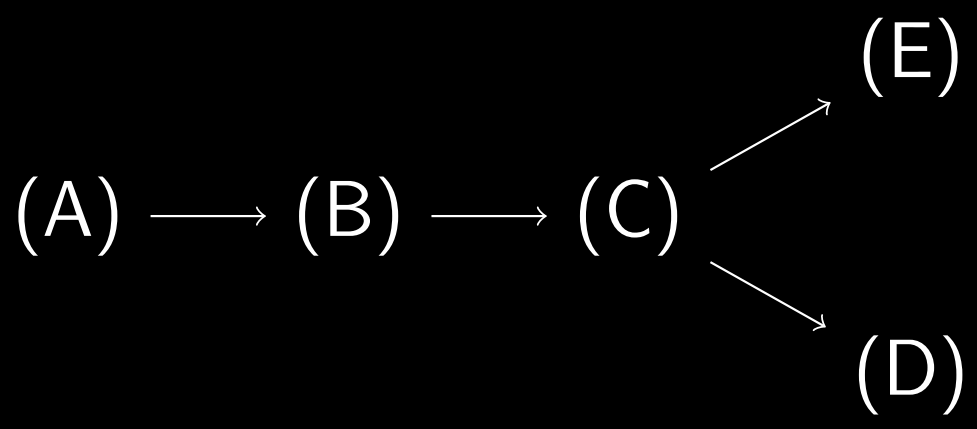
attempt 4: centralized

$$
\begin{tikzcd}
(\text{otherlocal}) & & (B) \arrow[r] & (C) \arrow[d] & \\
(\text{server}) & (A) \arrow[d] & (B) \arrow[u] & (C) \arrow[d] & \\
(\text{mylocal}) & (A) \arrow[r] & (B) \arrow[u] & (C) \arrow[r] & (D)
\end{tikzcd}
$$

attempt 4: centralized



attempt 4: centralized

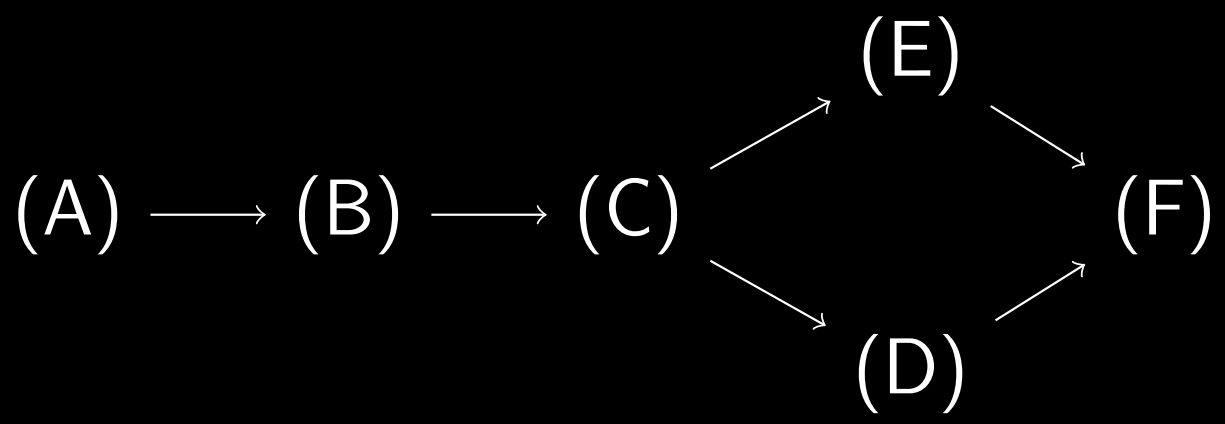


attempt 4: centralized

$$\begin{tikzcd}
 & & (E) \\
(A) \arrow[r] & (B) \arrow[r] & (C) \arrow[ur] \arrow[dr] \\
 & & (D)
\end{tikzcd}$$

"branching"

attempt 4: centralized



attempt 4: centralized

$$
\begin{tikzcd}
 & & & (E) \arrow[dr] & \\
(A) \arrow[r] & (B) \arrow[r] & (C) \arrow[ur] \arrow[dr] & & (F) \\
 & & & (D) \arrow[ur] &
\end{tikzcd}
$$

"merging"

attempt 4: centralized

this system sucks

attempt 4: centralized

this system sucks

- can't work without network access

attempt 4: centralized

this system sucks

- can't work without network access
- can't really version control scratchwork

attempt 4: centralized

this system sucks

- can't work without network access
- can't really version control scratchwork
- Linus says it sucks

attempt 4: centralized

this system sucks

- can't work without network access
- can't really version control scratchwork
- Linus says it sucks

The branching/merging stuff is cool, though.

attempt 4: centralized

this system sucks

- can't work without network access
- can't really version control scratchwork
- Linus says it sucks

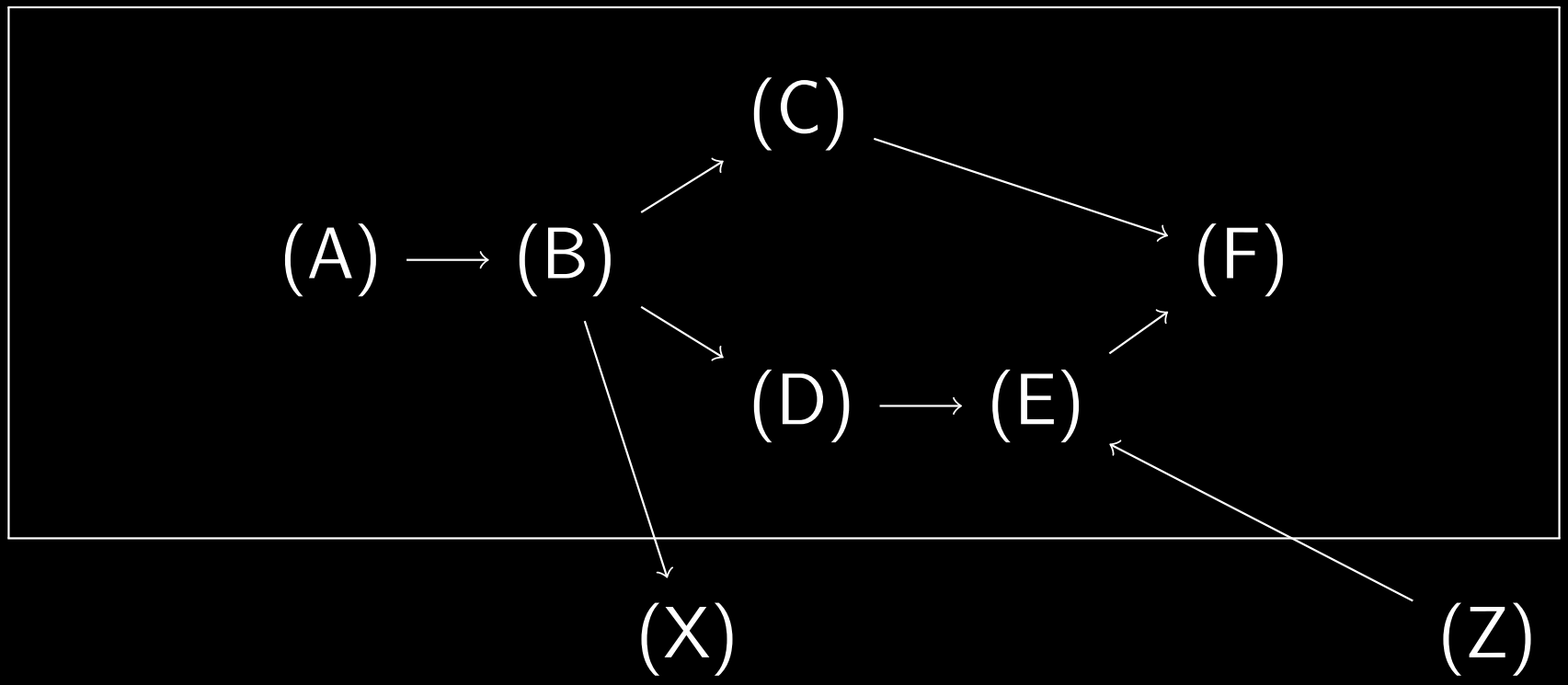
The branching/merging stuff is cool, though.

What if we could do that locally...

attempt 5: distributed

attempt 5: distributed

my local copy



other people's copies

attempt 5: distributed

this system rocks

attempt 5: distributed

this system rocks

there are a number of DVCSs:

attempt 5: distributed

this system rocks

there are a number of DVCSs:

git

attempt 5: distributed

this system rocks

there are a number of DVCSs:

git

mercurial

attempt 5: distributed

this system rocks

there are a number of DVCSs:

git

mercurial

bazarr

attempt 5: distributed

this system rocks

there are a number of DVCSs:

git

mercurial

bazarr

etc

attempt 5: distributed

this system rocks

there are a number of DVCSs:

git

mercurial

bazarr

etc

if you don't use git, fine, but use something distributed!

git pros and cons

git pros

git pros

- distributed

git pros

- distributed (which is done by others)

git pros

- distributed (which is done by others)
- integrity

git pros

- distributed (which is done by others)
- integrity
- crazy fast

git pros

- distributed (which is done by others)
- integrity
- crazy fast
- growing popularity

git pros

- distributed (which is done by others)
- integrity
- crazy fast
- growing popularity
- F/OSS

git pros

- distributed (which is done by others)
- integrity
- crazy fast
- growing popularity
- F/OSS (of course)

git pros

- distributed (which is done by others)
- integrity
- crazy fast
- growing popularity
- F/OSS (of course)
- github.com

git pros

- distributed (which is done by others)
- integrity
- crazy fast ⟵ main selling point
- growing popularity
- F/OSS (of course)
- github.com

git cons

git cons

- intended for *nix, not quite as fast on windows

git cons

- intended for *nix, not quite as fast on windows (mercurial is faster on windows)

git cons

- intended for *nix, not quite as fast on windows (mercurial is faster on windows)
- uses many languages internally

git cons

- intended for *nix, not quite as fast on windows (mercurial is faster on windows)
- uses many languages internally (C, bash, ruby, python, et al.)

git cons

- intended for *nix, not quite as fast on windows (mercurial is faster on windows)
- uses many languages internally (C, bash, ruby, python, et al.)
- considered one of the harder VCSs to learn

git cons

- intended for *nix, not quite as fast on windows (mercurial is faster on windows)
- uses many languages internally (C, bash, ruby, python, et al.)
- considered one of the harder VCSs to learn
  - half the difficulty can be remedied by learning the weird commands/syntax choices

git cons

- intended for *nix, not quite as fast on windows (mercurial is faster on windows)
- uses many languages internally (C, bash, ruby, python, et al.)
- considered one of the harder VCSs to learn
  - half the difficulty can be remedied by learning the weird commands/syntax choices
  - other half can be remedied by understanding what git is doing internally

git internrals

crypto hash functions

crypto hash functions

take some data, do maths on it, get something else

crypto hash functions

take some data, do maths on it, get something else

can't infer anything about original data from hash

crypto hash functions

take some data, do maths on it, get something else

can't infer anything about original data from hash

extremely difficult to create data with a specific hash as goal

crypto hash functions

git uses the sha1 hash

crypto hash functions

git uses the sha1 hash

"hello world" $\longrightarrow$
22596363b3de40b06f981fb85d82312e8c0ed511

crypto hash functions

git uses the sha1 hash

"hello world" $\longrightarrow$
22596363b3de40b06f981fb85d82312e8c0ed511

"hello worl**f**" $\longrightarrow$
d2552a097179b18fee3439bd1148100085c91e7f

crypto hash functions

uses:

crypto hash functions

uses:

verify two files are the same

crypto hash functions

uses:

verify two files are the same

verify a file hasn't been altered

typical filesystem

files:

typical filesystem

files:

“contains” data

typical filesystem

files:

“contains” data

directories:

typical filesystem

files:

“contains” data

directories:

“contain” files and other directories

typical filesystem

files:

“contains” data

directories:

“contain” files and other directories

symlinks (like shortcuts)

typical filesystem

files:

“contains” data

directories:

“contain” files and other directories

symlinks (like shortcuts)

“point” to files and directories

git is like a filesystem

yo dawg, i herd you like filesystems, so we put a filesystem in your filesystem so you can manage files while you manage files

blobs

blob $\approx$ file

blobs

blob $\approx$ file

- conceptually equivalent to a file in a filesystem

blobs

$$\text{blob} \approx \text{file}$$

- conceptually equivalent to a file in a filesystem
- contains data

blobs

$$\text{blob} \approx \text{file}$$

- conceptually equivalent to a file in a filesystem
- contains data
- compressed

blobs

$$\text{blob} \approx \text{file}$$

- conceptually equivalent to a file in a filesystem
- contains data
- compressed
- filename is sha1sum of contents

blobs

cfd971...	size
slides for a presentation on git. use pdflatex to compile. may require several packages, such as beamer.	

trees

tree $\approx$ directory

trees

$$\text{tree} \approx \text{directory}$$

- conceptually equivalent to a folder/directory in a filesystem

trees

$$\text{tree} \approx \text{directory}$$

- conceptually equivalent to a folder/directory in a filesystem
- contains list of blobs and trees

trees

tree $\approx$ directory

- conceptually equivalent to a folder/directory in a filesystem
- contains list of blobs and trees
  - permissions; **only executable or not executable**

trees

tree $\approx$ directory

- conceptually equivalent to a folder/directory in a filesystem
- contains list of blobs and trees
  - permissions; **only executable or not executable**
  - object type (blob or tree)

trees

tree $\approx$ directory

- conceptually equivalent to a folder/directory in a filesystem
- contains list of blobs and trees
  - permissions; **only executable or not executable**
  - object type (blob or tree)
  - sha1 hash of object (actual name on file, AND verification it's not been changed)

trees

tree $\approx$ directory

- conceptually equivalent to a folder/directory in a filesystem
- contains list of blobs and trees
  - permissions; **only executable or not executable**
  - object type (blob or tree)
  - sha1 hash of object (actual name on file, AND verification it's not been changed)
  - in-project filename of object

trees

5a9286... size

100644	blob	5d2cf1...	gittalk.tex
040000	tree	bb4277...	images
100644	blob	cfd972...	README
100755	blob	548981...	example.py

trees

5a9286... size

100644	blob	5d2cf1...	gittalk.tex
100644	tree	bb4277...	images

bb4277... size

100644	blob	3d5baa...	torvalds.jpg
100644	blob	acfd94...	notpr0n.jpg

100644	blob	cfd972...	README
100755	blob	548981...	example.py

commit

commit $\approx$ iterative backup

commit

commit $\approx$ iterative backup

- conceptually equivalent to an iterative backup

commit

commit $\approx$ iterative backup

- conceptually equivalent to an iterative backup
- effectively a snapshot of your project at a given point in time

commit

commit $\approx$ iterative backup

- conceptually equivalent to an iterative backup
- effectively a snapshot of your project at a given point in time
- with a bit of history and other metadata

commit

commit $\approx$ iterative backup

- conceptually equivalent to an iterative backup
- effectively a snapshot of your project at a given point in time
- with a bit of history and other metadata
  - pointer(s) to parent commits

commit

commit $\approx$ iterative backup

- conceptually equivalent to an iterative backup
- effectively a snapshot of your project at a given point in time
- with a bit of history and other metadata
  - pointer(s) to parent commits
  - author: person who wrote the changes since the last commit

commit

commit $\approx$ iterative backup

- conceptually equivalent to an iterative backup
- effectively a snapshot of your project at a given point in time
- with a bit of history and other metadata
  - pointer(s) to parent commits
  - author: person who wrote the changes since the last commit
  - committer: person who made the commit itself

commit

commit $\approx$ iterative backup

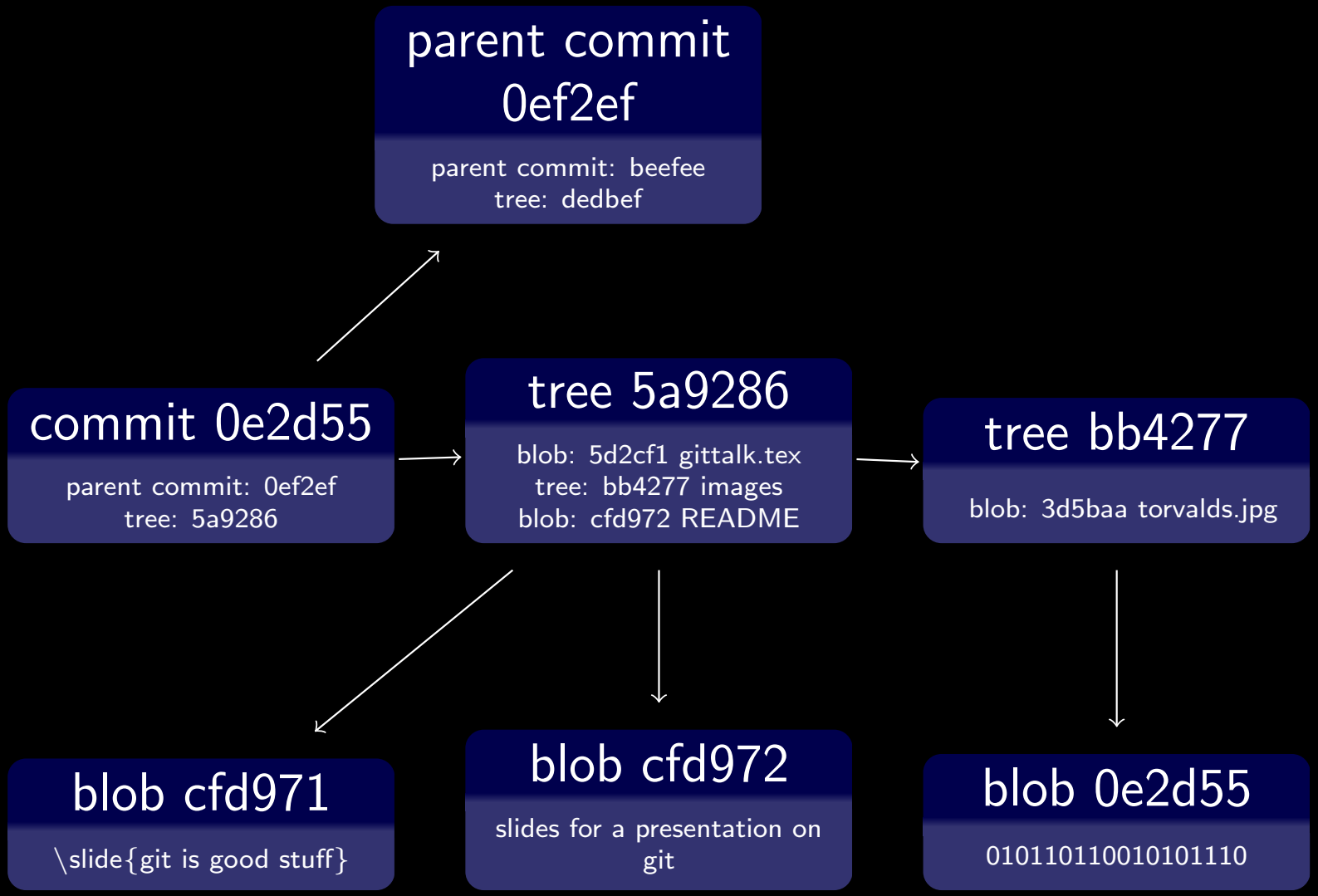
- conceptually equivalent to an iterative backup
- effectively a snapshot of your project at a given point in time
- with a bit of history and other metadata
  - pointer(s) to parent commits
  - author: person who wrote the changes since the last commit
  - committer: person who made the commit itself
  - comment: information about this commit; eg, what changed since last commit? why have the commit at all?

commit

0e2d55...	size

tree	5a9286...
parent	9f10c0...
author	paradigm
committer	paradigm
comment	reworked \slide, added arrows

map


parent commit 0ef2ef
parent commit: beefee
tree: dedbef
commit 0e2d55
parent commit: 0ef2ef
tree: 5a9286
tree 5a9286
blob: 5d2cf1 gittalk.tex
tree: bb4277 images
blob: cfd972 README
tree bb4277
blob: 3d5baa torvalds.jpg
blob cfd971
\slide{git is good stuff}
blob cfd972
slides for a presentation on git
blob 0e2d55
010110110010101110

trees and blobs

now that you understand trees and blobs

trees and blobs

now that you understand trees and blobs

you can forget about them

trees and blobs

now that you understand trees and blobs

you can forget about them

commits, heads and tags are where the action is at

heads/branches

head $\approx$ symlink to iterative backup

heads/branches

head $\approx$ symlink to iterative backup

typically a pointer to a commit (one exception)

heads/branches

head $\approx$ symlink to iterative backup

typically a pointer to a commit (one exception)

generally how you reference a commit that is actively considered/worked on

heads/branches

head $\approx$ symlink to iterative backup

typically a pointer to a commit (one exception)

generally how you reference a commit that is actively considered/worked on

"master" is usually the primary head

heads/branches

head $\approx$ symlink to iterative backup

typically a pointer to a commit (one exception)

generally how you reference a commit that is actively considered/worked on

"master" is usually the primary head

HEAD is a head pointing to the current head (often "master")

heads/branches

head $\approx$ symlink to iterative backup

typically a pointer to a commit (one exception)

generally how you reference a commit that is actively considered/worked on

"master" is usually the primary head

HEAD is a head pointing to the current head (often "master")

HEAD is like a symlink to where the next backup will go

heads/branches

HEAD
↓
master
↓
(A)

heads/branches

HEAD

↓

master

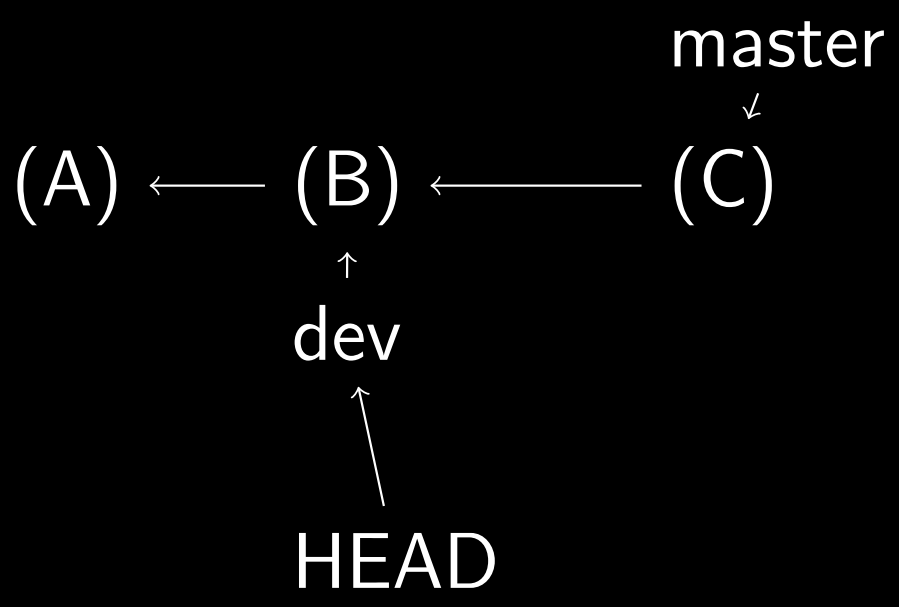
↙

(A) ⟵ (B)

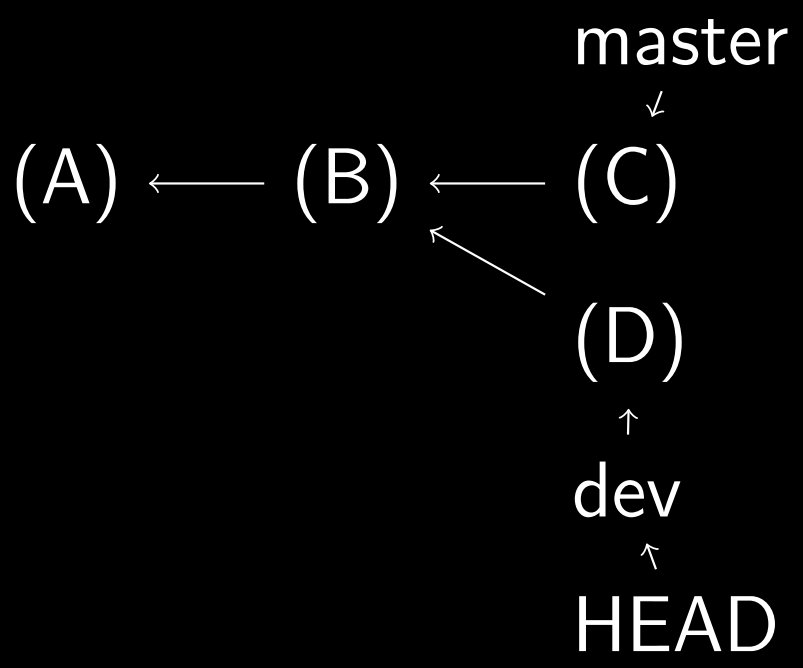
heads/branches

HEAD
↓
master
↓
(A) ⟵ (B) ⟵ (C)

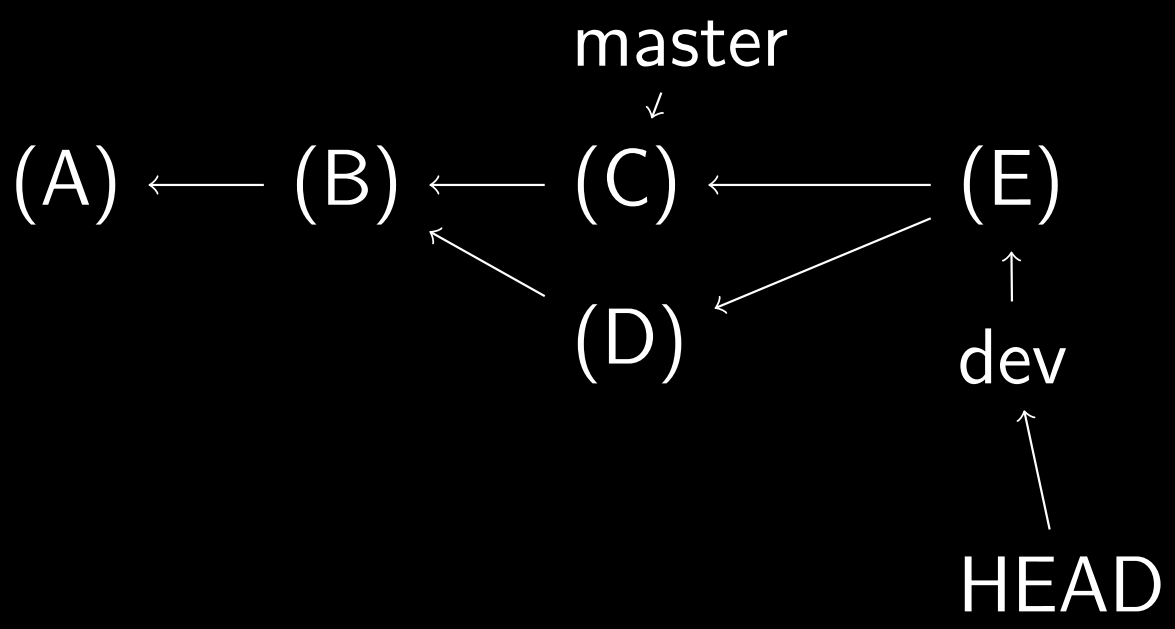
heads/branches



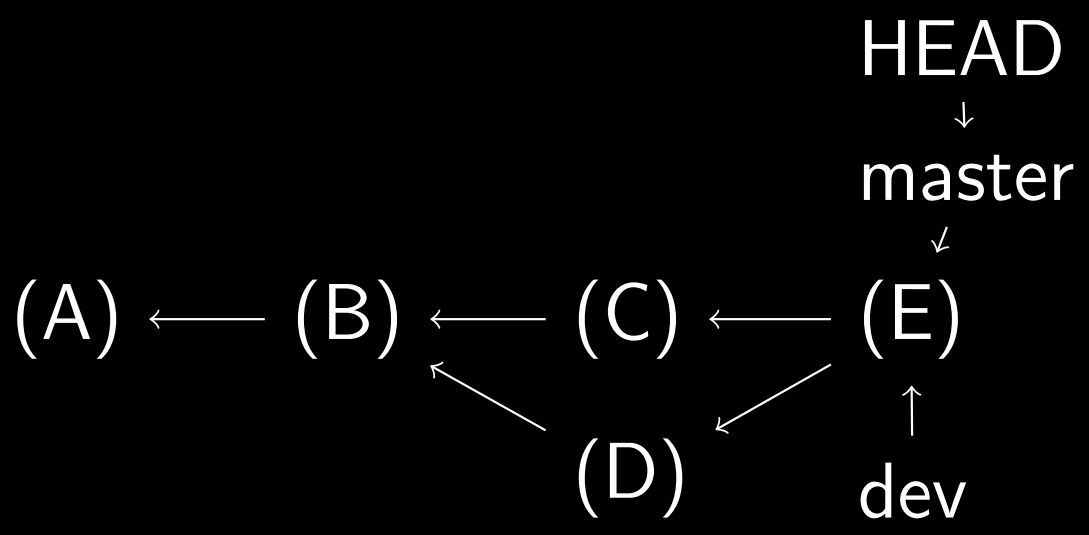
heads/branches



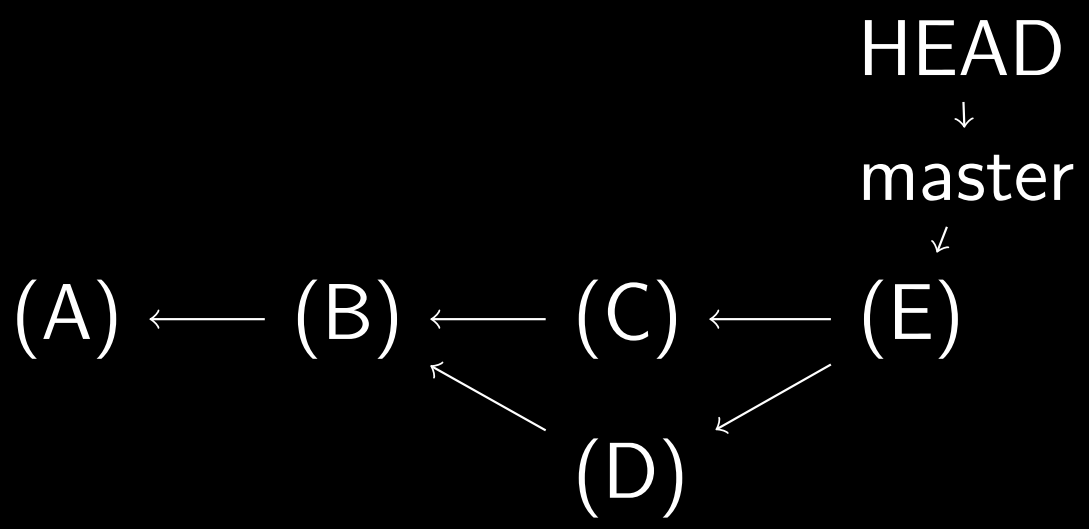
heads/branches



heads/branches



heads/branches



tags

tags are pointers to commits

tags

tags are pointers to commits

these are more for archival purposes, such as releases

tags

tags are pointers to commits

these are more for archival purposes, such as releases

e.g.: “v2.2.4”

tags

tags are pointers to commits

these are more for archival purposes, such as releases

e.g.: “v2.2.4”

can have an optional gpg signiture

summary

blobs $\approx$ files

summary

blobs $\approx$ files

trees $\approx$ directories, point to blobs and trees

summary

blobs $\approx$ files

trees $\approx$ directories, point to blobs and trees

commit $\approx$ iterative backups

summary

blobs $\approx$ files

trees $\approx$ directories, point to blobs and trees

commit $\approx$ iterative backups

heads $\approx$ symlinks to recently changed or soon to be changed backups

summary

blobs $\approx$ files

trees $\approx$ directories, point to blobs and trees

commit $\approx$ iterative backups

heads $\approx$ symlinks to recently changed or soon to be changed backups

tags $\approx$ symlinks to significant archival backups

three states of being

working directory

the copy of a files and directories you can edit

working directory

the copy of a files and directories you can edit

“checked out” copy

working directory

the copy of a files and directories you can edit

“checked out” copy

everything in the project’s directory other than the “.git” directory

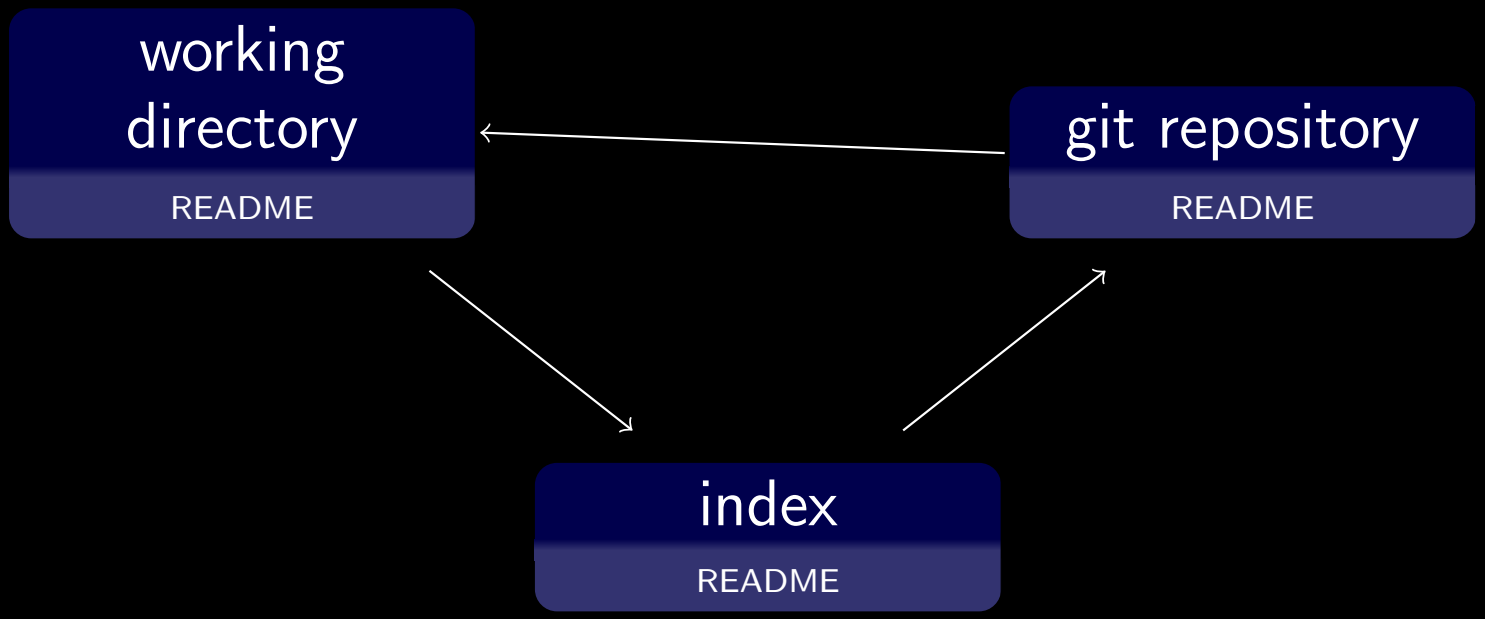
staged/indexed/cached

the state files are in which are about to be commited

git directory/repository

things in here are managed by git

life-cycle of a file



basic commands

command overview

git comes with roughly one million-billion commands

command overview

git comes with roughly one million-billion commands

most are “plumbing”

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

rest are “porcelain”

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

rest are “porcelain” - feel free to use

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

rest are “porcelain” - feel free to use

the command “git” can call all the other commands

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

rest are “porcelain” - feel free to use

the command “git” can call all the other commands

eg: “git add” calls “git-add”

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

rest are “porcelain” - feel free to use

the command “git” can call all the other commands

eg: “git add” calls “git-add”

feel free to use space or dash, either is fine

command overview

git comes with roughly one million-billion commands

most are “plumbing” - not for direct human consumption

rest are “porcelain” - feel free to use

the command “git” can call all the other commands

eg: “git add” calls “git-add”

feel free to use space or dash, either is fine

man git-foo for help

play along time!

if you've got git installed, feel free to play along

git-init

initializes a new git repository

git-init

initializes a new git repository

```
git-init /home/paradigm/osc_git_talk/
```

git-init

initializes a new git repository

```
git-init /home/paradigm/osc_git_talk/
```

```
git-init .
```

git-add

adds current version of file or directory to staging area/index/cache

git-add

adds current version of file or directory to staging area/index/cache

if you alter the file, the old version is still staged - **not** the newest version!

git-add

adds current version of file or directory to staging area/index/cache

if you alter the file, the old version is still staged - **not** the newest version!

```
git add README
```

git-add

adds current version of file or directory to staging area/index/cache

if you alter the file, the old version is still staged - **not** the newest version!

```
git add README
```

can add file to .gitignore ignore from future git-adds

git-rm

removes files or directories from staging area and working directory

git-rm

removes files or directories from staging area and working directory

just deleting file without telling git doesn't remove it from the staging area or being commited

git-rm

removes files or directories from staging area and working directory

just deleting file without telling git doesn't remove it from the staging area or being commited

```
git rm README
```

git-mv

renames or moves file or directory

git-mv

renames or moves file or directory

similar to git-rm, got to tell git for it to know to remove old place/name of the file

git-mv

renames or moves file or directory

similar to git-rm, got to tell git for it to know to remove old place/name of the file

not strictly necessary, since git uses hashes

git-mv

renames or moves file or directory

similar to git-rm, got to tell git for it to know to remove old place/name of the file

not strictly necessary, since git uses hashes

```
git mv README INSTALL
```

four pre-commit states of being

untracked: in the working directory, ignored by git

four pre-commit states of being

untracked: in the working directory, ignored by git

unmodified: unchanged from last copy in the git repository; staging and commiting won't do anything

four pre-commit states of being

untracked: in the working directory, ignored by git

unmodified: unchanged from last copy in the git repository; staging and commiting won't do anything

modified: previously tracked by git, changed since last commit, but not staged. Automatically staged with `git commit -a`

four pre-commit states of being

untracked: in the working directory, ignored by git

unmodified: unchanged from last copy in the git repository; staging and commiting won't do anything

modified: previously tracked by git, changed since last commit, but not staged. Automatically staged with `git commit -a`

staged: ready to be commited

git-status

shows current status of files in working directory, index, merge situation

git-status

shows current status of files in working directory, index, merge situation

```
git status
```

git-commit

create a commit from the staged files

git-commit

create a commit from the staged files

-a flag automatically git-add's files that have been modified or deleted. will **not** git-add new files.

git-commit

create a commit from the staged files

-a flag automatically git-add's files that have been modified or deleted. will **not** git-add new files.

prompts for comment with default editor, or in-line with -m flag

git-commit

create a commit from the staged files

-a flag automatically git-add's files that have been modified or deleted. will **not** git-add new files.

prompts for comment with default editor, or in-line with -m flag

–mmend: change last commit

git-commit

create a commit from the staged files

-a flag automatically git-add's files that have been modified or deleted. will **not** git-add new files.

prompts for comment with default editor, or in-line with -m flag

–mmend: change last commit

```
git commit --amend -a -m "fixed crashing on start bug"
```

git-log

shows log of the commits.

git-log

shows log of the commits.

–graph

git-log

shows log of the commits.

–graph

–pretty=oneline

git-log

shows log of the commits.

–graph

–pretty=oneline

–pretty=format:"%h - %an, %ar : %s"

git-log

shows log of the commits.

–graph

–pretty=oneline

–pretty=format:"%h - %an, %ar : %s"

–since=2.weeks

git-log

shows log of the commits.

–graph

–pretty=oneline

–pretty=format:"%h - %an, %ar : %s"

–since=2.weeks

```
git log --graph --pretty=oneline
```

git-clone

clones a repository

git-clone

clones a repository

```
git clone
http://opensource.osu.edu/git/yanovich/phenny
```

referring to a commit directly

- via head (eg: HEAD, master)

referring to a commit directly

- via head (eg: HEAD, master)
- via sha1 hash, or unique partial hash

referring to a commit directly

- via head (eg: HEAD, master)
- via sha1 hash, or unique partial hash
- (get from git-log)

referring to a commit directly

- via head (eg: HEAD, master)
- via sha1 hash, or unique partial hash
- (get from git-log)
- via tag

referring to a commit with offset

- branch@{time}, eg: `master@{yesterday}`

referring to a commit with offset

- branch@{time}, eg: `master@{yesterday}`
- branch@{N}, N commits back

referring to a commit with offset

- branch@{time}, eg: `master@{yesterday}`
- branch@{N}, N commits back
- commit^N, Nth parent (for merges)

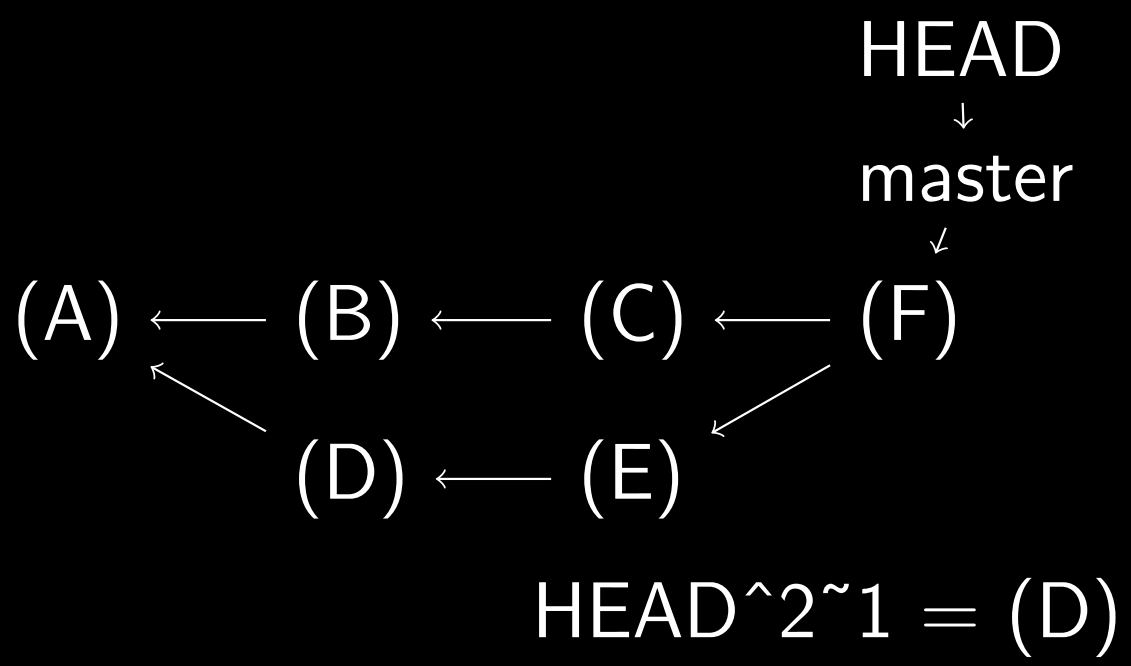
referring to a commit with offset

- branch@{time}, eg: `master@{yesterday}`
- branch@{N}, N commits back
- commit^N, Nth parent (for merges)
- commit~N, Nth grandparent

referring to a commit with offset

- branch@{time}, eg: `master@{yesterday}`
- branch@{N}, N commits back
- commit^N, Nth parent (for merges)
- commit~N, Nth grandparent
- can combine these: `master@{yesterday}~3^^^`

referring to a commit with offset



git-show

show information on blob, tree, tag or commit

git-show

show information on blob, tree, tag or commit

```
git show de03ee~3
```

git-reset

–mixed (default) resets index. undo a git-add

git-reset

–mixed (default) resets index. undo a git-add

–hard resets working directory and index. undo a change in the working directory.

git-reset

–mixed (default) resets index. undo a git-add

–hard resets working directory and index. undo a change in the working directory.

```
git reset .gittalk.swp HEAD
```

git-reset

–mixed (default) resets index. undo a git-add

–hard resets working directory and index. undo a change in the working directory.

```
git reset .gittalk.swp HEAD
```

```
git reset --hard HEAD
```

range of commits

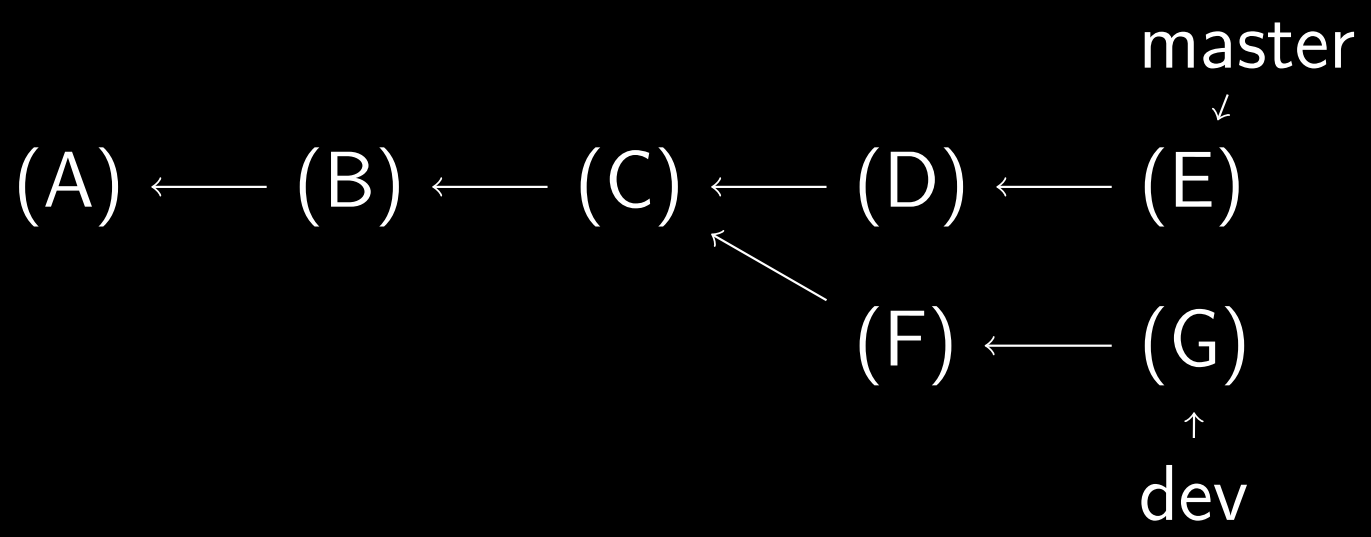
double dot: what is reachable from second but not first?

range of commits

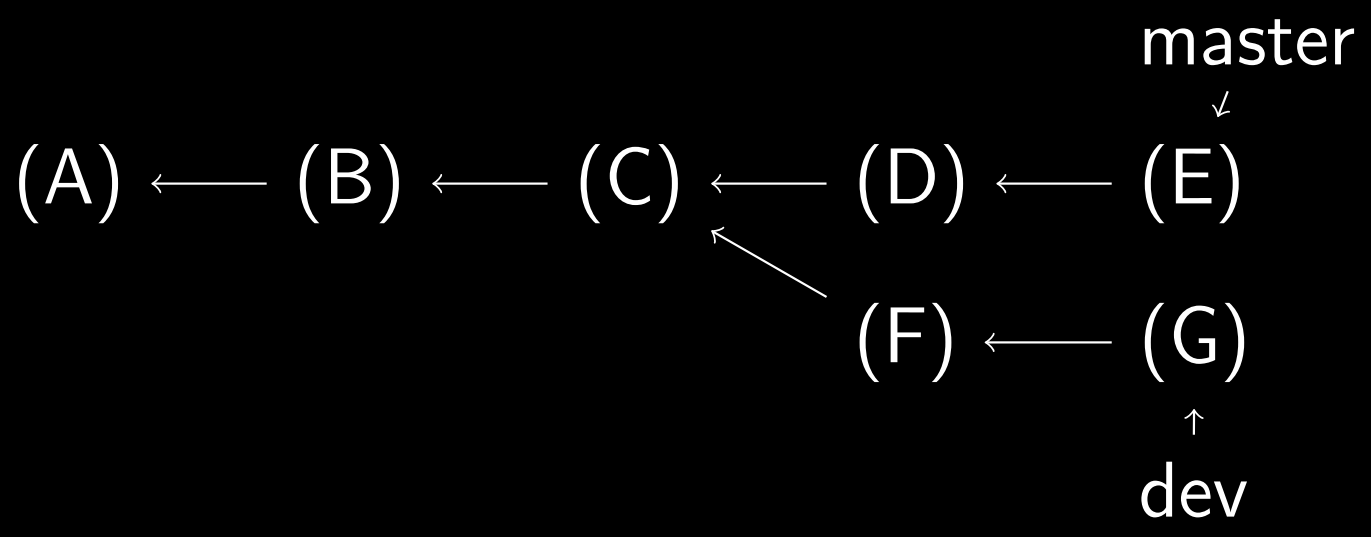
double dot: what is reachable from second but not first?

triple dot: what commits are reachable from either but not both?

range of commits

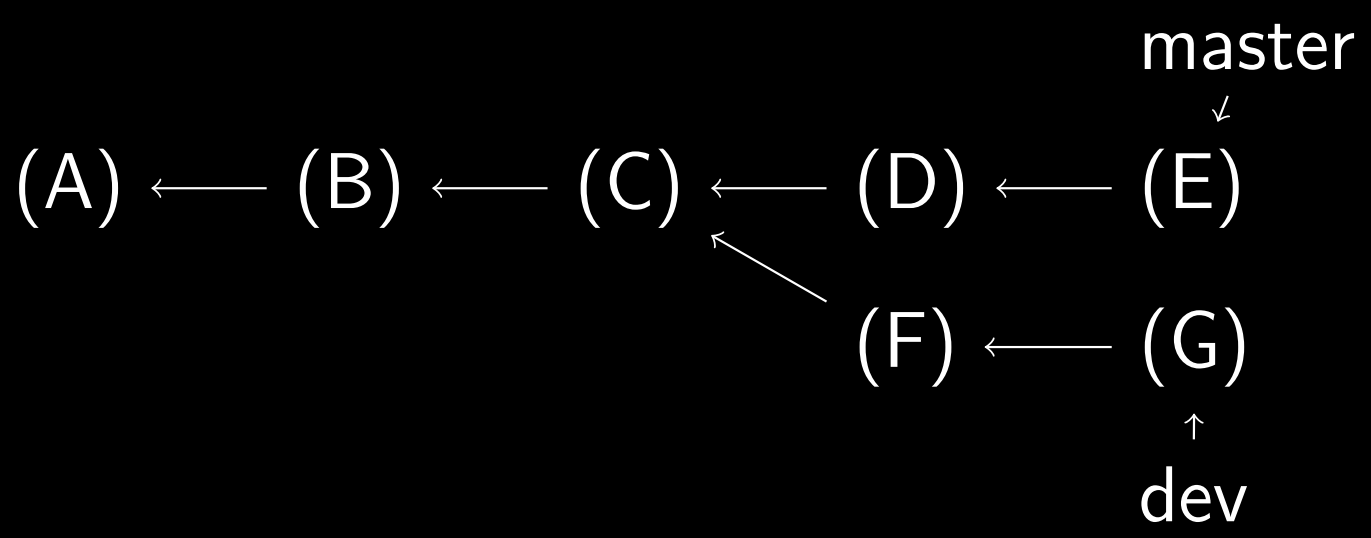


range of commits



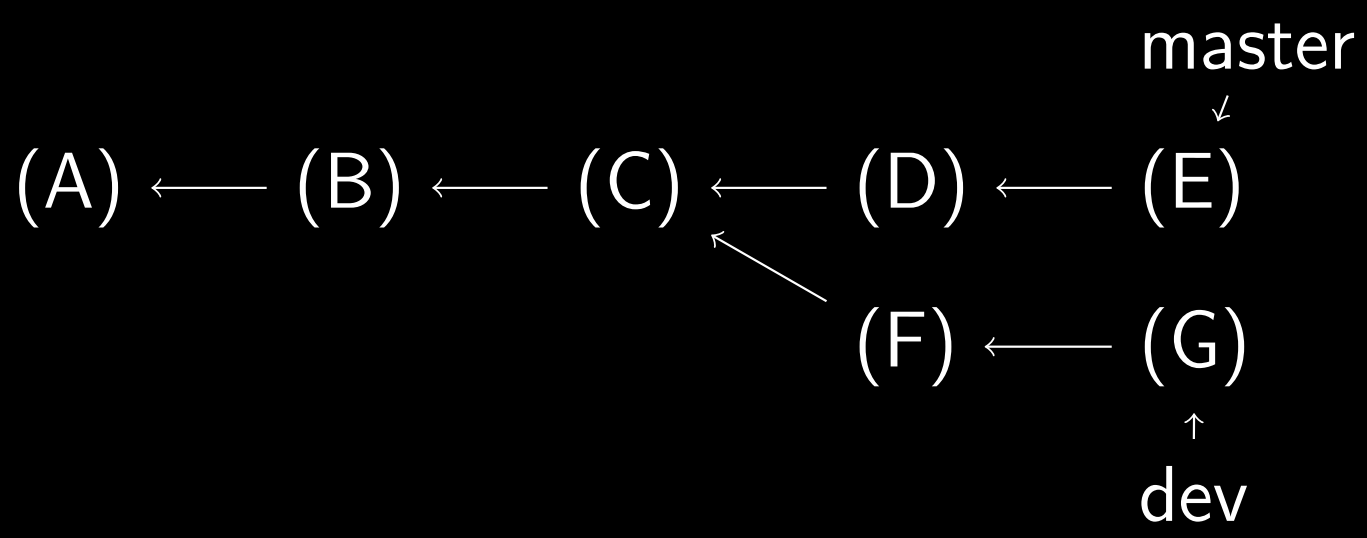
`git log master~3..master`: (C), (D)

range of commits



`git log master~3..master`: (C), (D)
`git log master..dev`: (G), (F)

range of commits



`git log master~3..master`: (C), (D)
`git log master..dev`: (G), (F)
`git log master...dev`: (D), (E), (F), (G)

git-diff

compare between working area, index, and commits

git-diff

compare between working area, index, and commits

no flag: show difference between modified and staged

git-diff

compare between working area, index, and commits

no flag: show difference between modified and staged

–cached: show difference between unmodified and staged (ie, staged)

git-diff

compare between working area, index, and commits

no flag: show difference between modified and staged

–cached: show difference between unmodified and staged (ie, staged)

can take files or commits as arguments

git-diff

compare between working area, index, and commits

no flag: show difference between modified and staged

–cached: show difference between unmodified and staged (ie, staged)

can take files or commits as arguments

```
git diff HEAD~2 HEAD
```

git-diff

compare between working area, index, and commits

no flag: show difference between modified and staged

–cached: show difference between unmodified and staged (ie, staged)

can take files or commits as arguments

```
git diff HEAD~2 HEAD
```

```
git diff --cached README
```

git-branch

no arguments: list heads

git-branch

no arguments: list heads

one argument: create new head

git-branch

no arguments: list heads

one argument: create new head

-d flag: deletes head

git-branch

no arguments: list heads

one argument: create new head

-d flag: deletes head

`git branch`

git-branch

no arguments: list heads

one argument: create new head

-d flag: deletes head

```
git branch
```

```
git branch dev
```

git-branch

no arguments: list heads

one argument: create new head

-d flag: deletes head

```
git branch
```

```
git branch dev
```

```
git branch -d dev
```

git-checkout

move HEAD to branch, changes working directory to branch's commit's files

git-checkout

move HEAD to branch, changes working directory to branch's commit's files

-b flag: git-branch foo && git-checkout foo

git-checkout

move HEAD to branch, changes working directory to branch's commit's files

-b flag: git-branch foo && git-checkout foo

```
git checkout dev
```

git-checkout

move HEAD to branch, changes working directory to branch's commit's files

-b flag: git-branch foo && git-checkout foo

```
git checkout dev
```

```
git checkout -b test
```

git-checkout

move HEAD to branch, changes working directory to branch's commit's files

-b flag: git-branch foo && git-checkout foo

```
git checkout dev
```

```
git checkout -b test
```

can be used to reset one file to a previous version:

git-checkout

move HEAD to branch, changes working directory to branch's commit's files

-b flag: git-branch foo && git-checkout foo

```
git checkout dev
```

```
git checkout -b test
```

can be used to reset one file to a previous version:

```
git checkout HEAD~3 README
```

git-merge

create a new commit by combining two (or more) branches

git-merge

create a new commit by combining two (or more) branches

```
git merge dev
```

branching visual

branching visual

HEAD
↓
master
↙
(A)

```
vi foo && git add foo && git commit
```

branching visual

HEAD

↓

master

↙

(A) ⟵ (B)

```
vi foo && git add foo && git commit
```

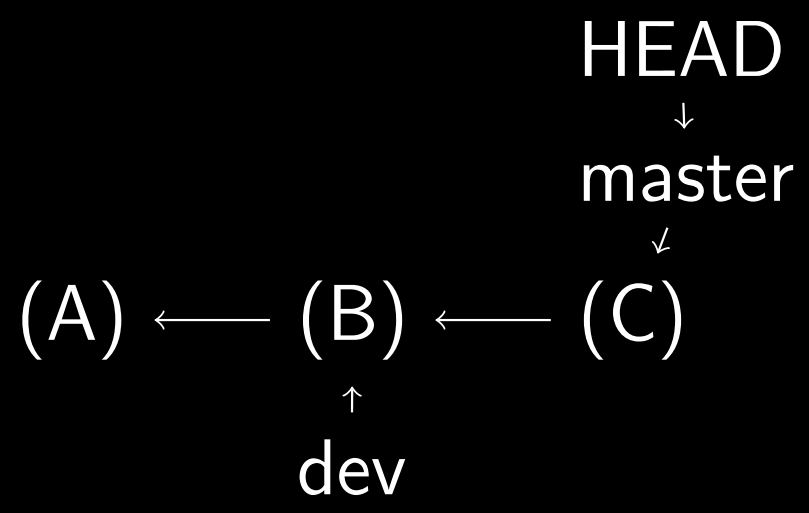
branching visual

HEAD
↓
master
↙

(A) ⟵ (B) ⟵ (C)

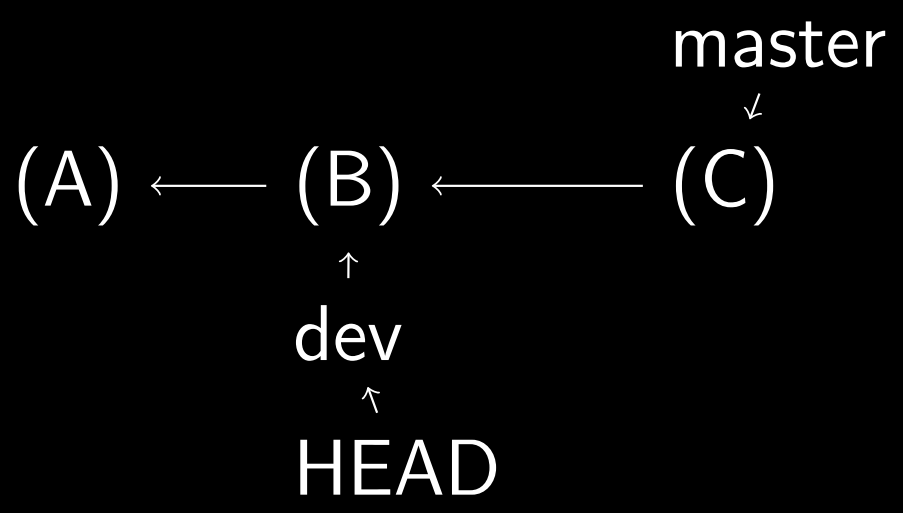
```
vi foo && git add foo && git commit
```

branching visual



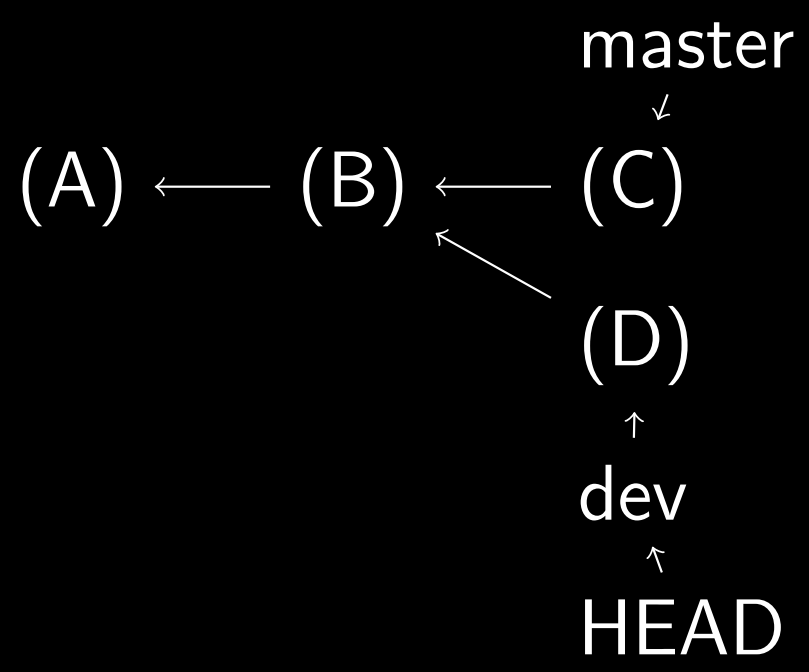

```
git branch dev HEAD~2
```

branching visual



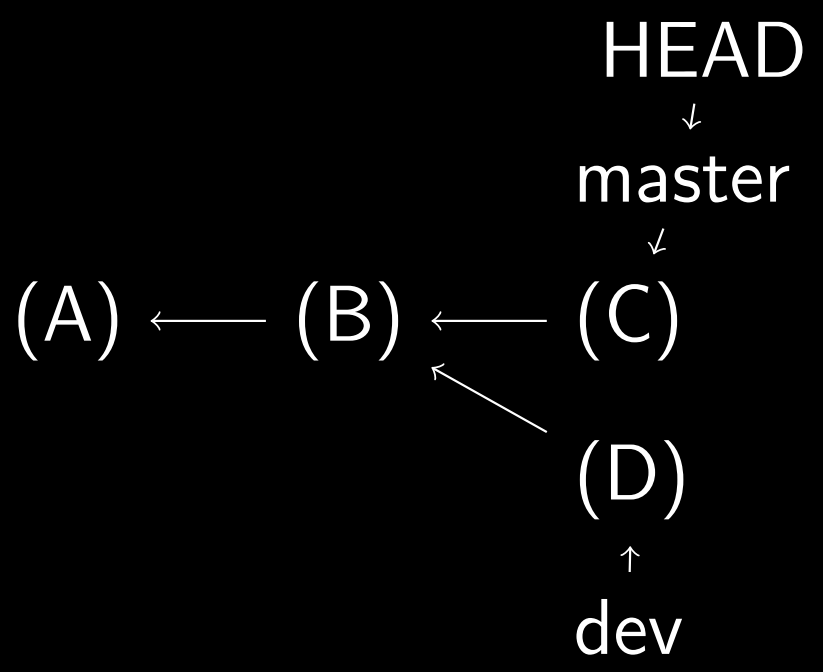

```
git checkout dev
```

branching visual



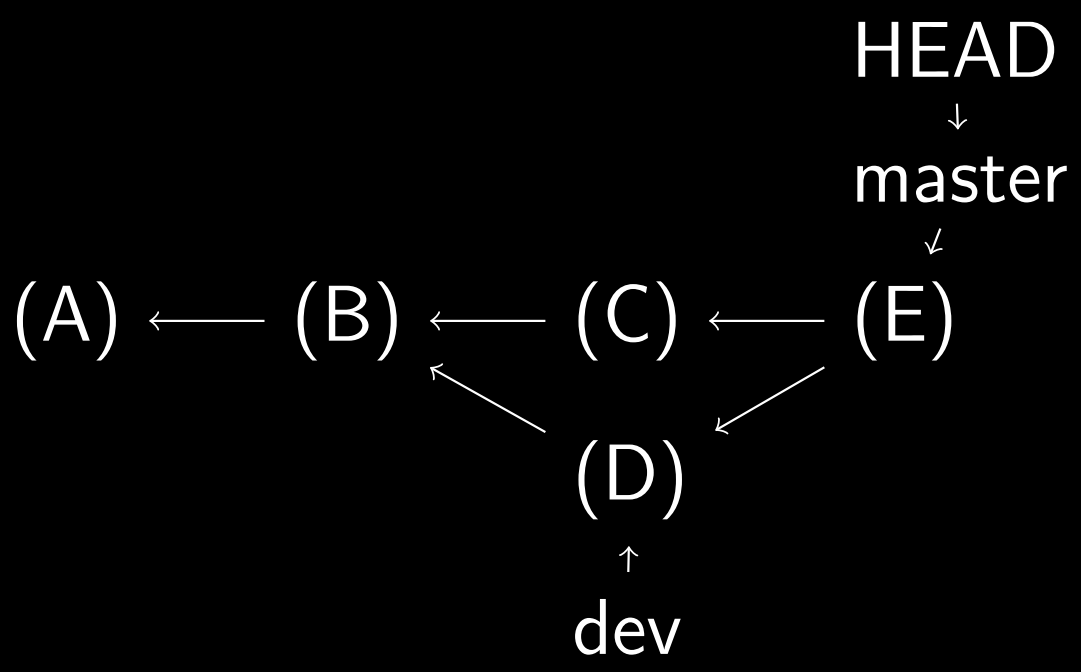

```
vi foo && git add foo && git commit
```

branching visual



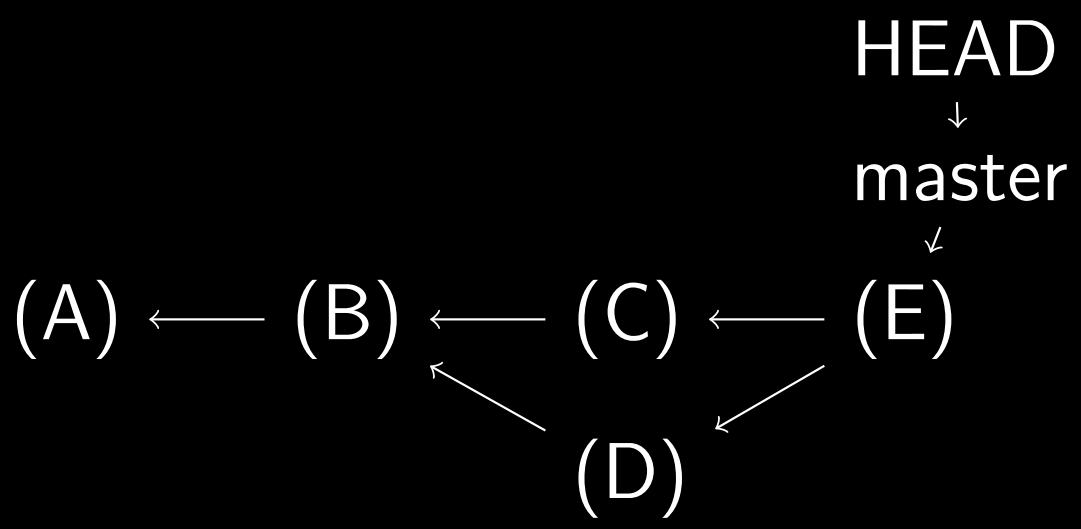

```
git checkout master
```

branching visual




```
git merge dev
```

branching visual




```
git branch -d dev
```

how merging works

go back in history to last shared commit

how merging works

go back in history to last shared commit

look at diffs along other branch

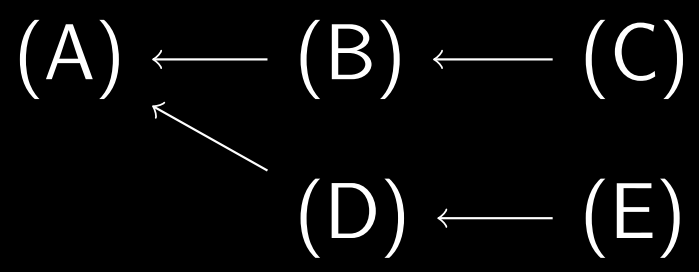
how merging works

go back in history to last shared commit

look at diffs along other branch

apply to current branch

how merging works




```
git checkout (C) && git merge (E)
```

how merging works

$$
\begin{tikzcd}
(A) & (B) \arrow[l] & (C) \arrow[l] \\
 & (D) \arrow[ul] & (E) \arrow[l]
\end{tikzcd}
$$

common ancestor (A)

how merging works

$$\begin{tikzcd}
(A) & (B) \arrow[l] & (C) \arrow[l] \\
 & (D) \arrow[ul] & (E) \arrow[l]
\end{tikzcd}$$

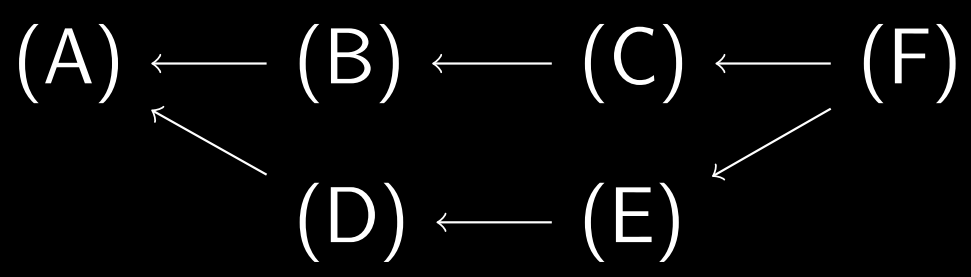
apply $\Delta(D)$ to (C)

how merging works

$$
\begin{tikzcd}
(A) & (B) \arrow[l] & (C) \arrow[l] \\
 & (D) \arrow[ul] & (E) \arrow[l]
\end{tikzcd}
$$

apply $\Delta(E)$ to (C)

how merging works



move current head to (F)

how merging works

if git gets confused when mergeing:

how merging works

if git gets confused when mergeing:

`git status` to see where the conflict is

how merging works

if git gets confused when mergeing:

`git status` to see where the conflict is

edit relevant file(s). $<<<$, $===$, and $>>>$ mark where the conflict is

how merging works

if git gets confused when mergeing:

`git status` to see where the conflict is

edit relevant file(s). $<<<$, $===$, and $>>>$ mark where the conflict is

once fixed, `git commit` to finish merger

how merging works

if git gets confused when mergeing:

`git status` to see where the conflict is

edit relevant file(s). $<<<$, $===$, and $>>>$ mark where the conflict is

once fixed, `git commit` to finish merger

or cancel with `git reset --hard HEAD`

branch workflows

Stable and dev branches

master

↙

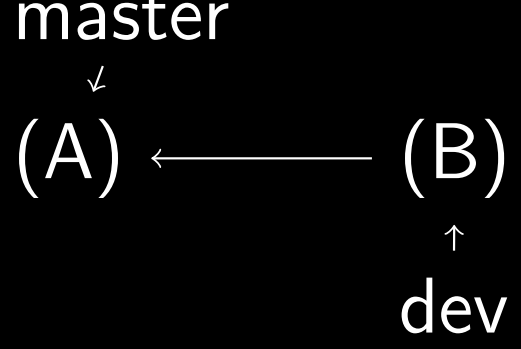
(A)

↑

dev

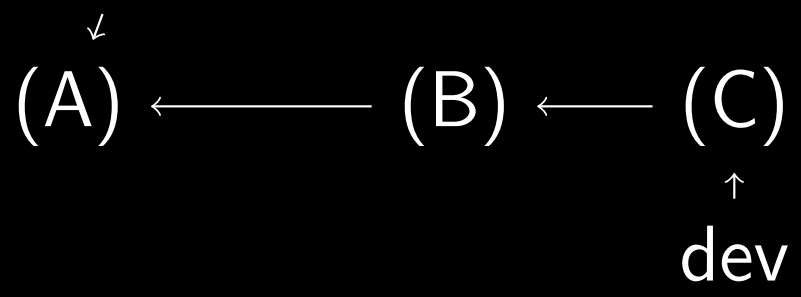
branch workflows

Stable and dev branches



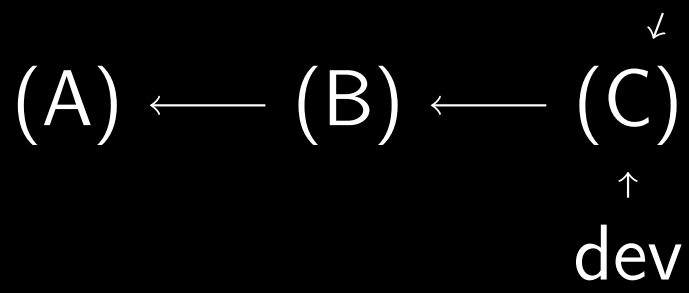
branch workflows

Stable and dev branches



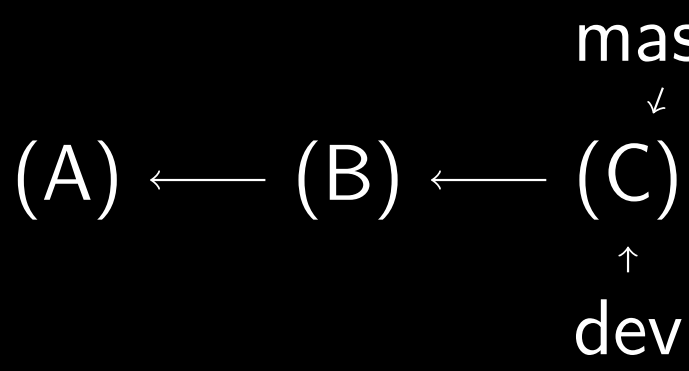
branch workflows

Stable and dev branches



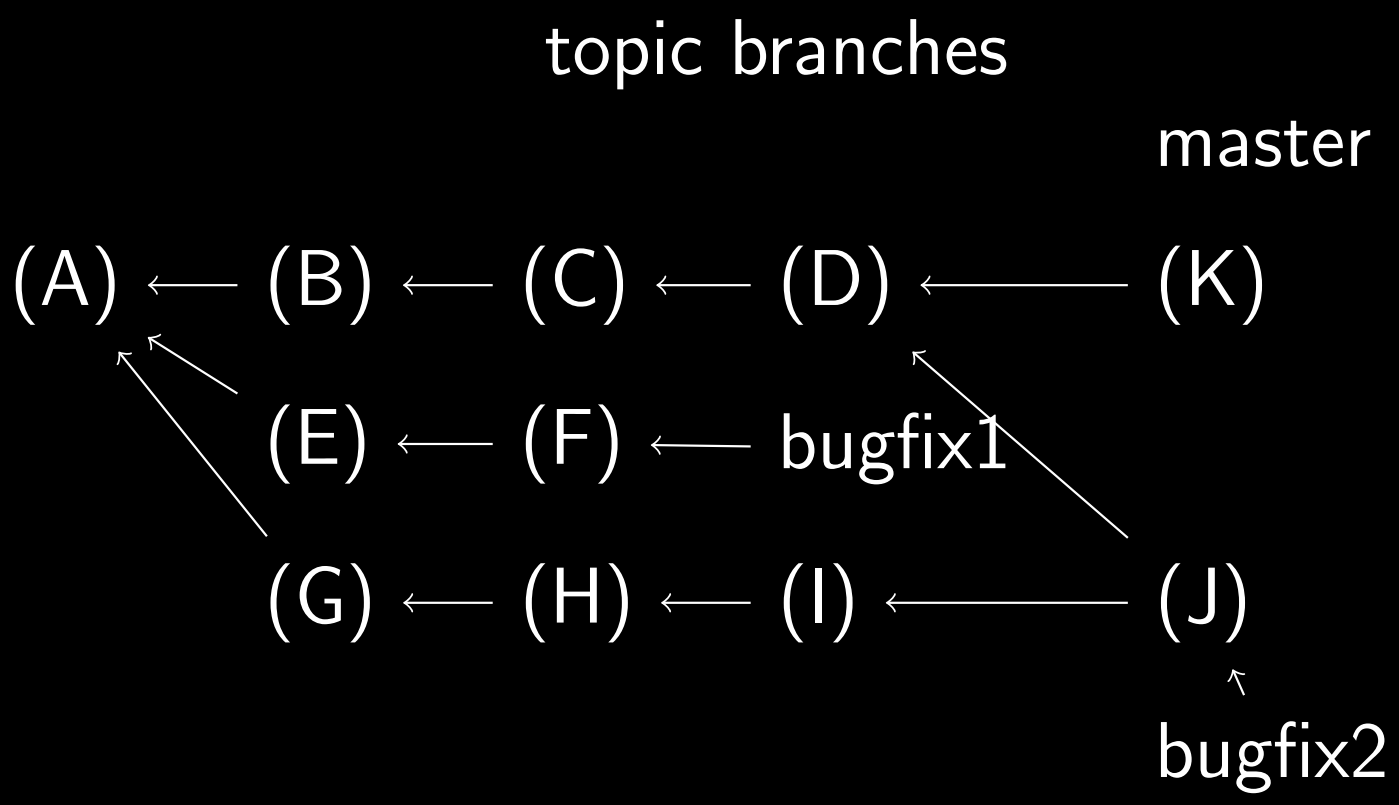
branch workflows

Stable and dev branches



"fast-forward merge"

branch workflows



remote branch

essentially just a head pointing to a head on someone else's computer, for tracking purposes

remote branch

essentially just a head pointing to a head on someone else's computer, for tracking purposes

automatically created with `git-clone`

remote branch

essentially just a head pointing to a head on someone else's computer, for tracking purposes

automatically created with `git-clone`

e.g.: origin/master

git-remote

manage tracked repositories

git-remote

manage tracked repositories

add: adds new tracked repository

git-remote

manage tracked repositories

add: adds new tracked repository

rm: deletes tracked repository

git-fetch

update information from a tracking branch

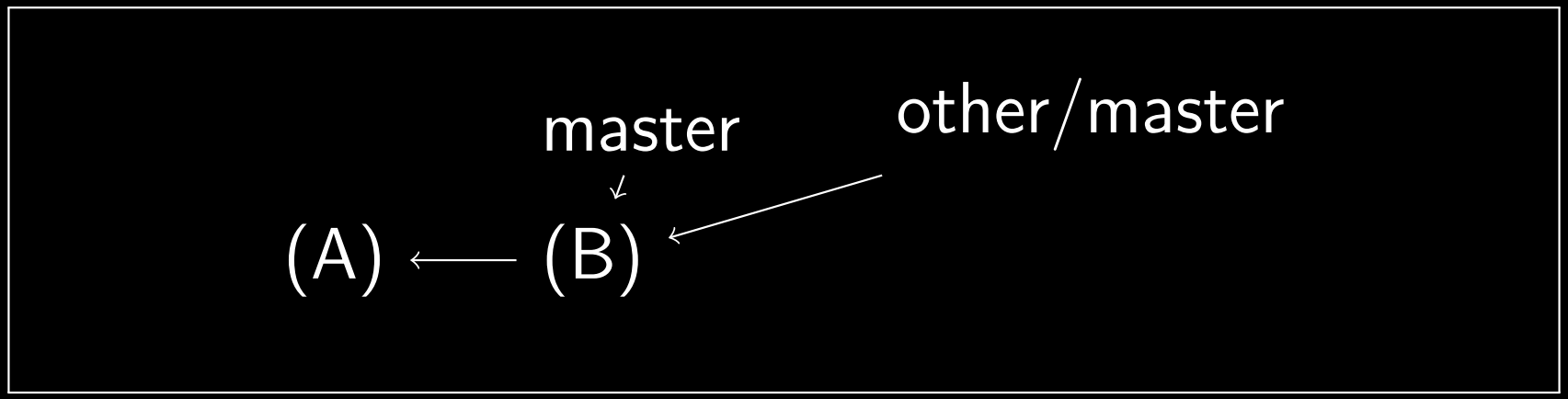
remote visual

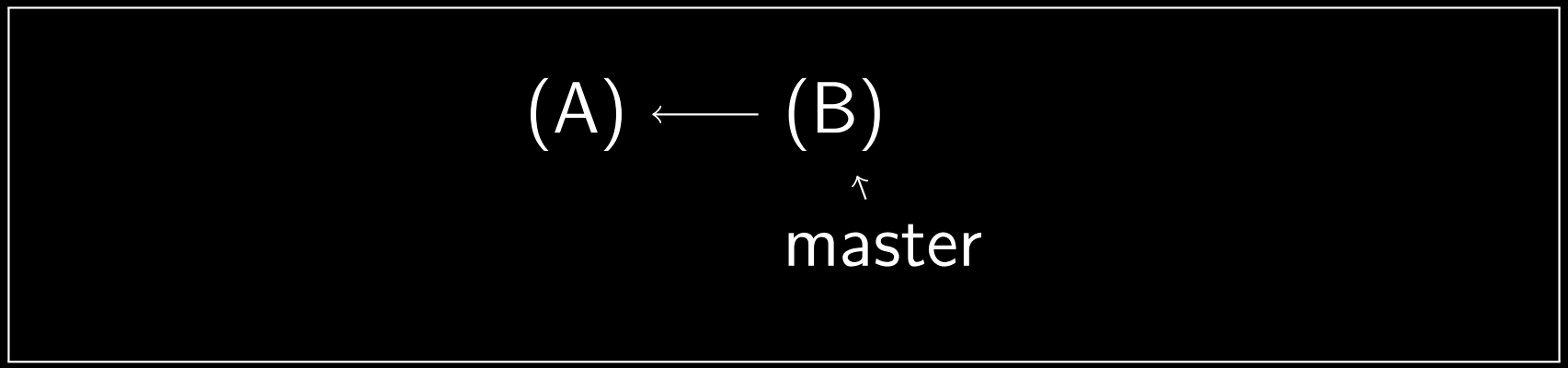
(A) ⟵ (B)

↖

master

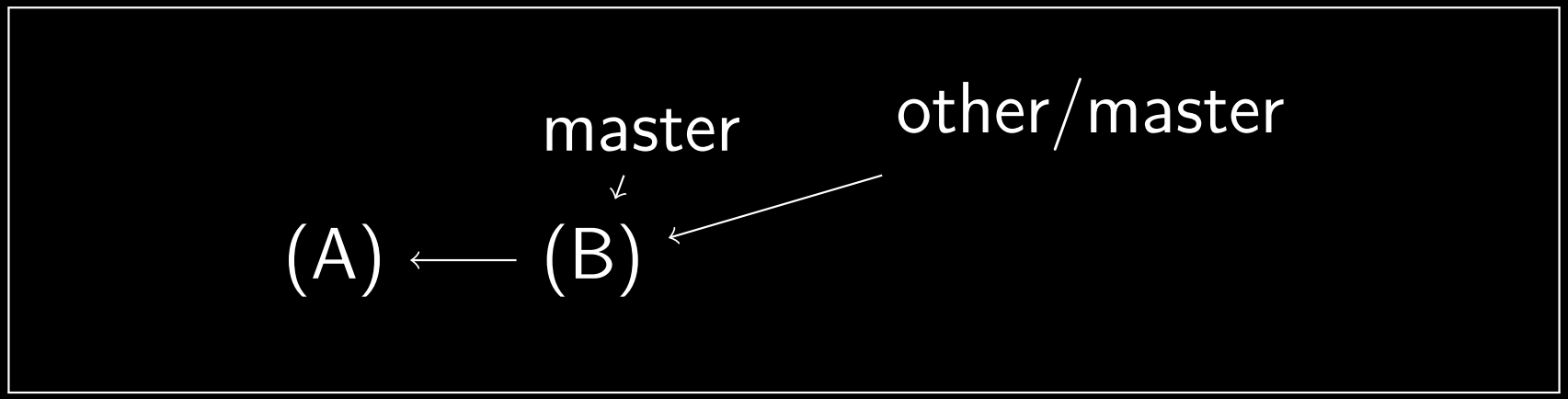
remote visual

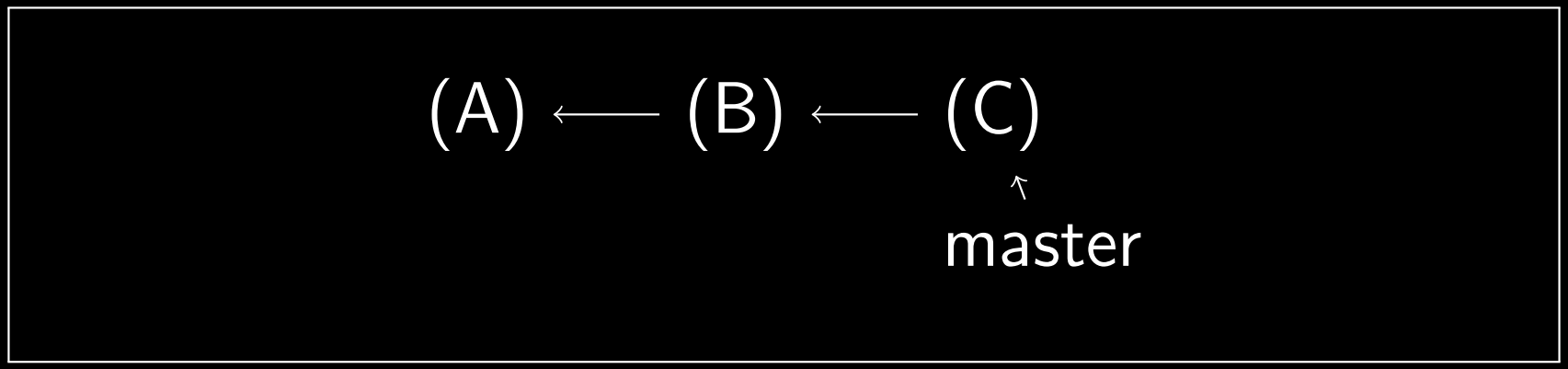





```
git clone foo.com/foo.git
```

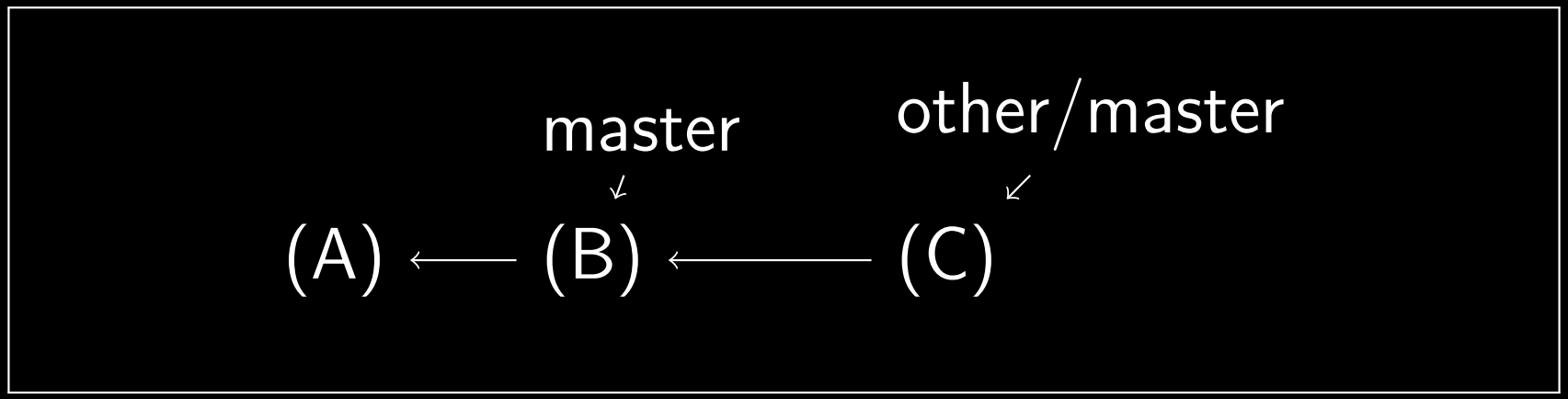
remote visual

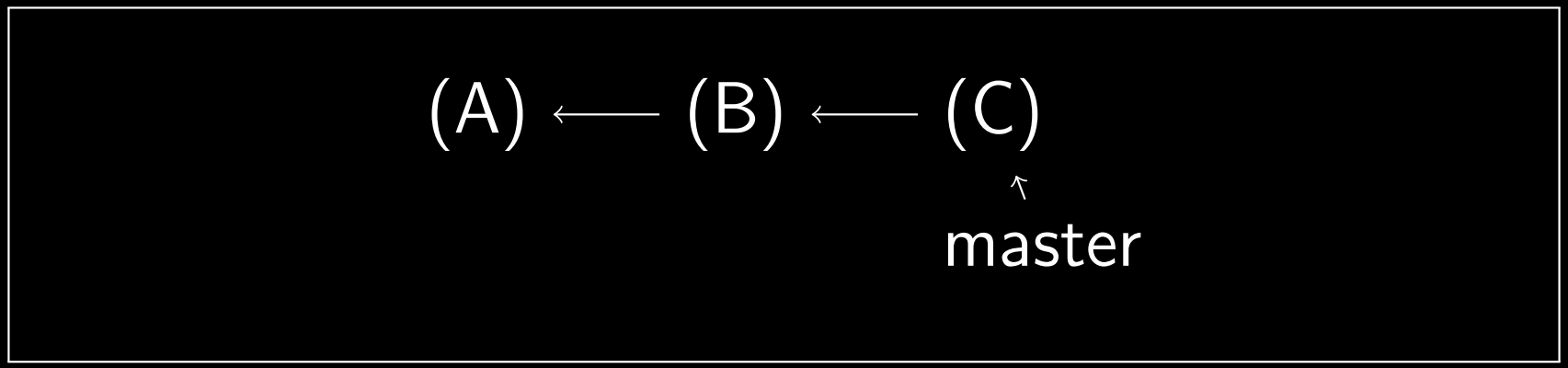




other guy does: `git commit`

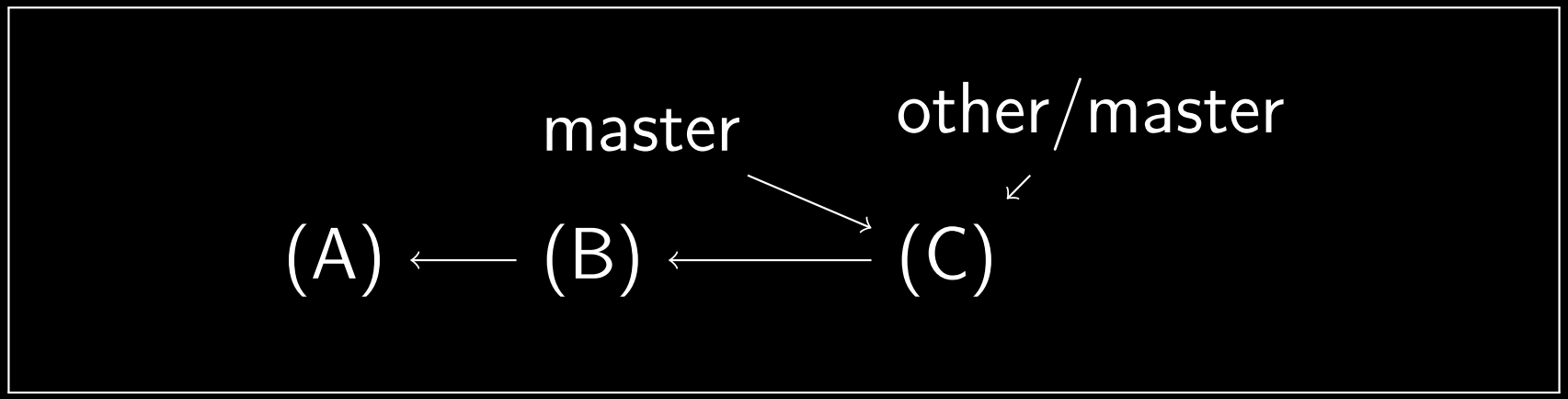
remote visual

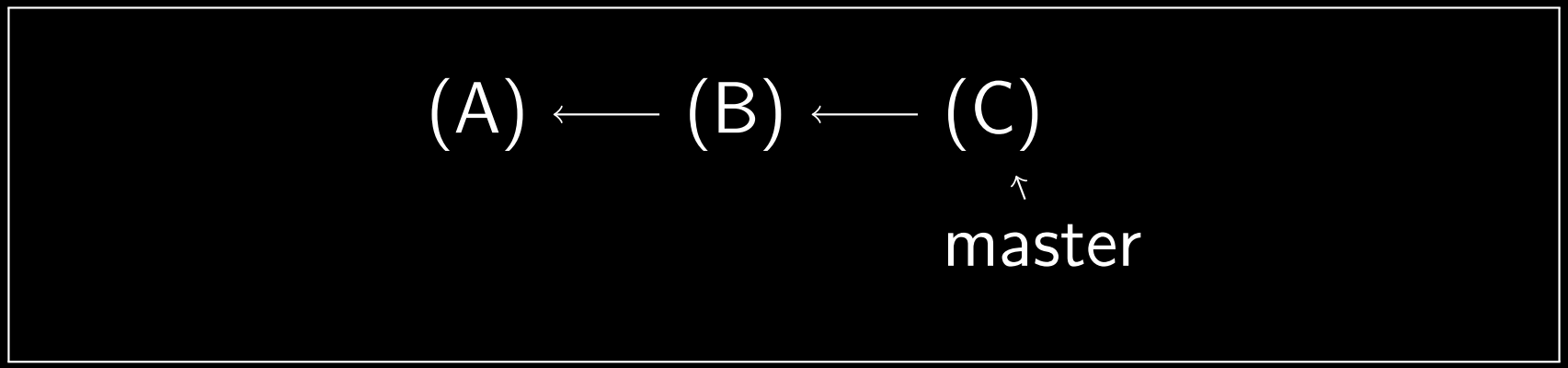





```
git fetch origin
```

remote visual






```
git merge origin master
```

git-pull

combines `git-fetch` and `git-merge` into one command

git-push

requests other computer to do a `git-pull` on your branch

git-push

requests other computer to do a `git-pull` on your branch

useful for a central repository

git-push

requests other computer to do a `git-pull` on your branch

useful for a central repository

most be up-to-date before pushing or will get error!

git-push

requests other computer to do a `git-pull` on your branch

useful for a central repository

most be up-to-date before pushing or will get error!

delete remote branch:
`git push origin :branchname`

git protocols

git can use the following protocols for fetch/push/pull:

git protocols

git can use the following protocols for fetch/push/pull:

local protocol

git protocols

git can use the following protocols for fetch/push/pull:

local protocol

ssh protocol

git protocols

git can use the following protocols for fetch/push/pull:

local protocol

ssh protocol

git protocol

git protocols

git can use the following protocols for fetch/push/pull:

local protocol

ssh protocol

git protocol

http(s) protocol

medium git

tags

tags are usually used to mark releases

tags

tags are usually used to mark releases

two types of tags: lightweight and annotated

tags

tags are usually used to mark releases

two types of tags: lightweight and annotated

lightweight: basically a frozen branch, little metadata

tags

tags are usually used to mark releases

two types of tags: lightweight and annotated

lightweight: basically a frozen branch, little metadata

annotated: lots of metadata, such as optional gpg key

git-tag

no argument/flags: list tags

git-tag

no argument/flags: list tags

one argument, no flags: create lightweight tag

git-tag

no argument/flags: list tags

one argument, no flags: create lightweight tag

-a flag: annontated

git-tag

no argument/flags: list tags

one argument, no flags: create lightweight tag

-a flag: annontated

-s flag: annontated and signed

git-config

```
git config --user.name "Daniel Thau"
```

git-config

```
git config --user.name "Daniel Thau"
    git config --core.editor vim
```

git-config

```
git config --user.name "Daniel Thau"
git config --core.editor vim
git config --global alias.co checkout
```

git-stash

```
git stash
```

git-stash

```
git stash
git stash list
```

git-stash

```
git stash
git stash list
git stash apply
```

git-stash

```
git stash
git stash list
git stash apply
git stash branch
```

git add -i

interactive menus for selecting what should be staged/indexed/cached

git add -i

interactive menus for selecting what should be staged/indexed/cached

potentially easier than running `git status` every thirty seconds

referring to trees or blobs

tree: commit^{tree}

referring to trees or blobs

tree: commit^{tree}

blob: commit:/path/to/file

git-grep

like grep, but can search through git's history

git-gc

compress history

git-gc

compress history

takes a while

git-fsck

does a number of consistency checks, just in case

git-fsck

does a number of consistency checks, just in case

often warns about “dangling objects” - don’t worry, not harmful

git-blame

shows who changed what on each line

git-blame

shows who changed what on each line

```
git blame README
```

git-format-patch

produces series of patch files in current directory

git-send-email

grabs patches from `git format-patch` and emails them

git-send-email

grabs patches from `git format-patch` and emails them

good when you've got a lot of patches

bare git repository

no working directory

bare git repository

no working directory

good for push/pull server

bare git repository

no working directory

good for push/pull server

`git --bare init`

advanced git

changing history

don't change history of something you've pushed or someone else has pulled.

changing history

don't change history of something you've pushed or someone else has pulled.

may be useful before a push/pull to clean things up and make it easier for others to parse

changing last commit

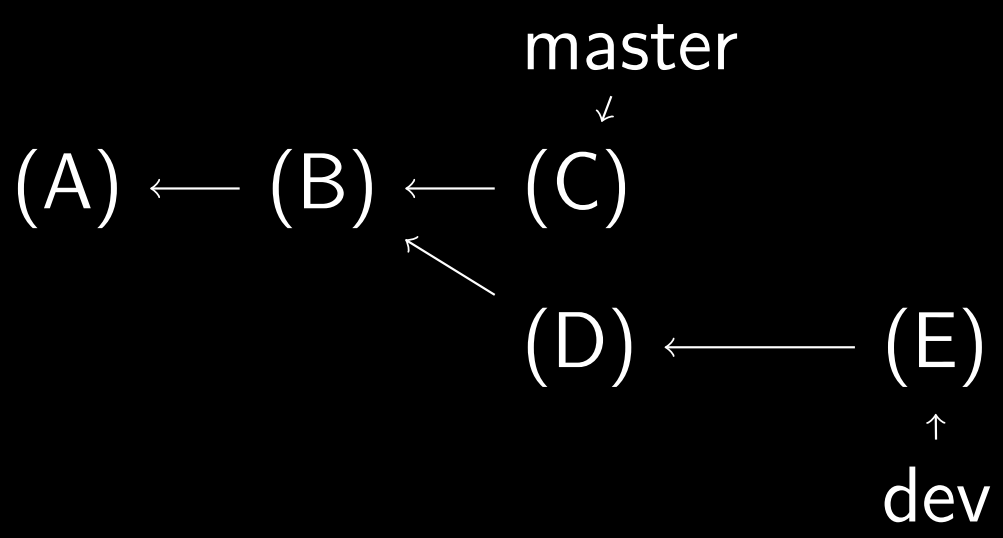
make changes in working directory

changing last commit

make changes in working directory

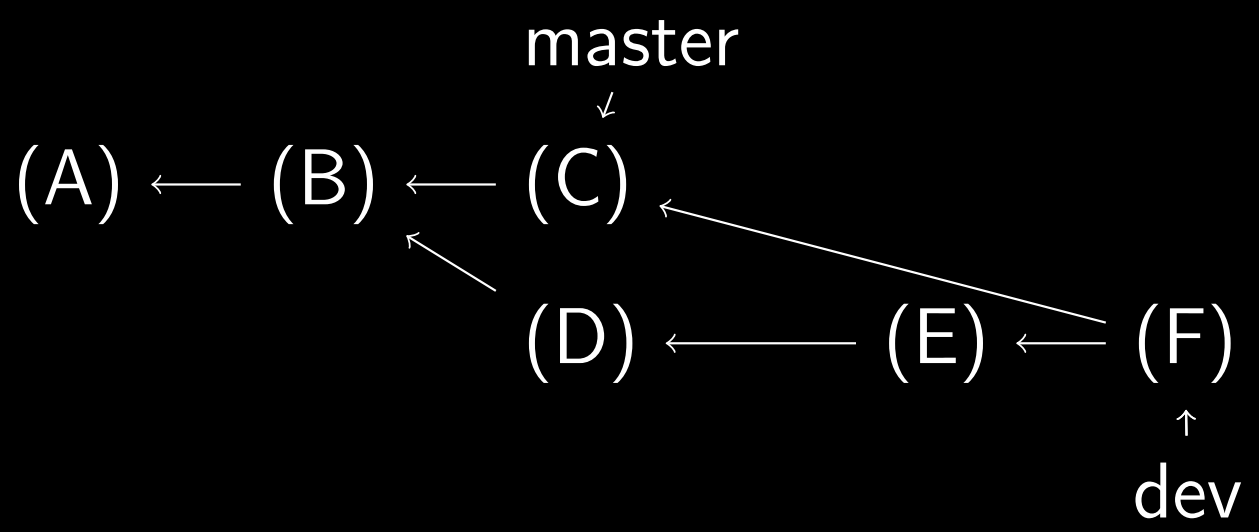
```
git commit --amend
```

git rebase



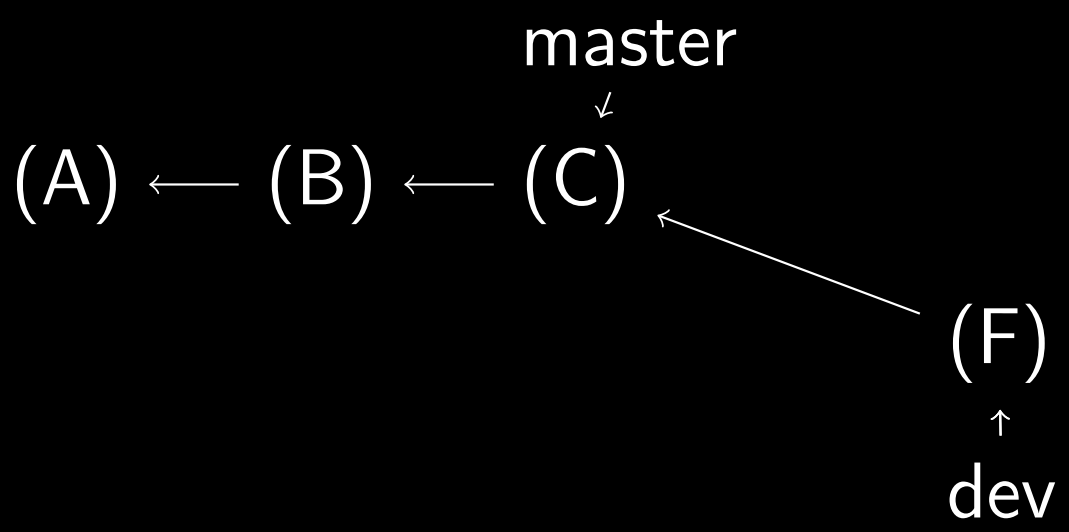

```
git checkout dev
```

git rebase




```
git merge master
```

git rebase




```
git rebase master
```

git rebase

if conflict arises:

git rebase

if conflict arises:

fix like a merge commit

git rebase

if conflict arises:

fix like a merge commit

`git add` fixed files

git rebase

if conflict arises:

fix like a merge commit

`git add` fixed files

`git rebase --continue`

git rebase

if conflict arises:

fix like a merge commit

`git add` fixed files

`git rebase --continue`

or you can always

git rebase

if conflict arises:

fix like a merge commit

`git add` fixed files

`git rebase --continue`

or you can always

`git rebase --abort`

git-rebase -i

alter between current commit and argument

git-rebase -i

alter between current commit and argument

```
git rebase -i HEAD~3
```

git-rebase -i

alter between current commit and argument

```
git rebase -i HEAD~3
```

will list commits thusly:

git-rebase -i

alter between current commit and argument

`git rebase -i HEAD~3`

will list commits thusly:

(action) (partial-sha1) (commit message)

git-rebase -i

alter between current commit and argument

`git rebase -i HEAD~3`

will list commits thusly:

(action) (partial-sha1) (commit message)

re-arrange commit order

git-rebase -i

alter between current commit and argument

`git rebase -i HEAD~3`

will list commits thusly:

(action) (partial-sha1) (commit message)

re-arrange commit order

change the (action)

git-rebase -i “pick”

(action)=“pick”: leave alone, continue with rebasing

git-rebase -i “squash”

(action)=“squash”: combine with previous commit

git-rebase -i “squash”

(action)=“squash”: combine with previous commit

will prompt for new commit comment

git-rebase -i “squash”

(action)=“squash”: combine with previous commit

will prompt for new commit comment

may have merge conflict - treat like normal merge conflicts

git-rebase -i “squash”

(action)=“squash”: combine with previous commit

will prompt for new commit comment

may have merge conflict - treat like normal merge conflicts

rebasing will continue

git-rebase -i “edit”

(action)=“edit”: change commit

git-rebase -i “edit”

(action)=“edit”: change commit

rebasing will pause at selected commit

git-rebase -i “edit”

(action)=“edit”: change commit

rebasing will pause at selected commit

edit files

git-rebase -i “edit”

(action)=“edit”: change commit

rebasing will pause at selected commit

edit files

`git add`

git-rebase -i “edit”

(action)=“edit”: change commit

rebasing will pause at selected commit

edit files

`git add`

`git commit --amend`

git-rebase -i “edit”

(action)=“edit”: change commit

rebasing will pause at selected commit

edit files

`git add`

`git commit --amend`

`git rebase --continue`

git-rebase -i “edit”

(action)=“edit”: change commit

rebasing will pause at selected commit

edit files

`git add`

`git commit --amend`

`git rebase --continue`

may have merge conflict - treat like normal merge conflicts

git-bisect

great for finding when a regression occurred

git-bisect

great for finding when a regression occurred

```
git bisect start
```

git-bisect

great for finding when a regression occurred

```
git bisect start
git bisect good foo
```

git-bisect

great for finding when a regression occurred

```
git bisect start
git bisect good foo
git bisect bad bar
```

git-bisect

great for finding when a regression occurred

`git bisect start`

`git bisect good foo`

`git bisect bad bar`

regression? `git bisect bad`

git-bisect

great for finding when a regression occurred

`git bisect start`

`git bisect good foo`

`git bisect bad bar`

regression? `git bisect bad`

good? `git bisect good`

git-bisect

great for finding when a regression occurred

`git bisect start`

`git bisect good foo`

`git bisect bad bar`

regression? `git bisect bad`

good? `git bisect good`

finished? git bisect reset

git hooks

handy scripts that are automatically run at certain times

git hooks

handy scripts that are automatically run at certain times

located in `.git/hooks`

git hooks

handy scripts that are automatically run at certain times

located in `.git/hooks`

git ignores those that end in `.sample`

git hooks

handy scripts that are automatically run at certain times

located in `.git/hooks`

git ignores those that end in `.sample`

comment explains details

git hooks

handy scripts that are automatically run at certain times

located in `.git/hooks`

git ignores those that end in `.sample`

comment explains details

edit to your heart's content

references

“understanding git conceptually”

references

“understanding git conceptually”

http://www.eecs.harvard.edu/~cduan/technical/git/

references

“understanding git conceptually”

`http://www.eecs.harvard.edu/~cduan/technical/git/`

“pro git”

references

“understanding git conceptually”

http://www.eecs.harvard.edu/~cduan/technical/git/

“pro git”

http://progit.org/book/

references

“understanding git conceptually”

http://www.eecs.harvard.edu/~cduan/technical/git/

“pro git”

http://progit.org/book/

“git community book”

references

“understanding git conceptually”

http://www.eecs.harvard.edu/~cduan/technical/git/

“pro git”

http://progit.org/book/

“git community book”

http://book.git-scm.com/