

Git: Version control

How do we track how code changes?

Joseph Hallett

January 29, 2024



What's this all about?

- ▶ You can't write all the code you need in a day.
- ▶ Sometimes you have to share code with other people.
 - ▶ Doing that thing where you number files and stick 'FINAL.docx' is stupid.

We need a mechanism to systematically track changes.

Luckily

Software Engineers solved this problem back in the 70s.

- ▶ (and perfected it in the 2000s)

For example

Suppose Alice goes and writes the following program:

```
public class Hello {
    public static void main(String[] args) {
        if (args.length == 0) {
            args = new String[1];
            args[0] = "World";
        }
        for (final var name : args)
            System.out.print("Hello_" + name + "!\\n");
    }
}
```

Later she updates it...

```
public class Hello2 {
    public static void main(String[] args) {
        if (args.length == 0) {
            args = new String[1];
            args[0] = "World";
        }
        for (final var name : args)
            System.out.println("Hello_" + name + "!");
    }
}
```

Whats changed?

A bad solution

We *could* go and track changes manually...

- ▶ Each version of the file has a different name with a number on the end
- ▶ Write a suite of tools for spotting what the differences in files are...

```
2c2
< public class Hello {
---
> public class Hello2 {
9c9
<         System.out.print("Hello_" + name + "! \n");
---
>         System.out.println("Hello_" + name + "!");
```

And we could store the diffs over time to keep a record of how things changed, and who changed what...

Don't work hard! Work Lazy!

Clearly managing source code like this is going to be a lot of manual work. But we're *computer scientists*... we can automate *anything*.

So lets do that!

- ▶ Write software to do all the management of software for you
- ▶ Let it keep track of who has changed what and when
- ▶ Let the programmer step in and fix things as a last resort

Git

This is *Linus Torvalds* he made Git (and Linux).

To make a change to the kernel:

- ▶ You take a copy of the code
- ▶ Do your work
- ▶ Email a diff to Linus
- ▶ And you'll have a discussion about it over email
 - ▶ See <https://lore.kernel.org/lkml>
- ▶ If he likes it Linus will apply it to the codebase

Git is designed to be a tool to help Linus do his job

- ▶ Not designed to be user friendly
- ▶ Worse is better
- ▶ Fast for working with plaintext files (source code)
- ▶ Works well with *huge* numbers of files
- ▶ Source code isn't that complex

This is *still* how the kernel gets developed!



Git is hard

There's a manual

GIT(1)

Git Manual

GIT(1)

NAME

git - the stupid content tracker

SYNOPSIS

```
git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
    [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
    [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
    [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
    [--super-prefix=<path>] [--config-env=<name>=<envvar>]
    <command> [<args>]
```

DESCRIPTION

Git is a fast, scalable, distributed revision control system with an unusually rich command **set** that provides both high-level operations and full access to internals.

See gittutorial(7) to get started, **then** see giteveryday(7) **for** a useful

There are also books

<https://git-scm.com/book/en/v2> The *official* Git book.

<https://ohshitgit.com> A guide for how to get out of silly situations in Git.

You have to practice for years for it to become comfortable

- ▶ See you in the labs

Okay lets get started!

To create a *Git repo* we can use the `git init` command:

```
mkdir tutorial  
cd tutorial  
git init
```

```
Initialized empty Git repository in /tmp/tutorial/.git/
```

```
ls -A
```

```
.git
```

```
git status
```

```
On branch main
```

```
No commits yet
```

```
nothing to commit (create/copy files and use "git add" to track)
```


Lets add some code

```
cat >hello.c <<EOF
#include <stdio.h>

int main(void) {
    printf("Hello, World\n");
    return 0;
}
EOF
git add hello.c
git status
```

On branch main

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)
new file: hello.c

Staging

At this point, the file `hello.c` is *staged* but it hasn't been *committed* yet.

When you *stage* a file:

- ▶ You're saying this will be part of a new commit
- ▶ You're adding the changes into Git's versioning
- ▶ But you're not saving anything
- ▶ Things can still change!

When you *commit*:

- ▶ Everything you've staged so far gets written into the history as a single change.
- ▶ With a note explaining it
- ▶ And your name associated with it
- ▶ Things *shouldn't* change
 - ▶ (technically they still can... but it gets harder)

Lets commit!

```
git commit -m 'Initial commit of the greeting program.  
Greets the user and then exits.'
```

```
[main (root-commit) 6f436f6] Initial commit of the greeting program.  
1 file changed, 6 insertions(+)  
create mode 100644 hello.c
```

```
[main (root-commit) 65a5a16] Initial commit of the greeting program.  
1 file changed, 6 insertions(+)  
create mode 100644 hello.c
```

Note

Sometimes when your on a new system you'll get a prompt to set your name and email... just follow the instructions provided. All Git commits need a name and an email address attributed to them.

```
git config --global user.name 'Joseph Hallett'  
git config --global user.email 'joseph.hallett@bristol.ac.uk'
```

Lets make some edits

```
ed hello.c <<EOS
3c
int main(int argc, char *argv[]) {
.
4c
    for (int i=0; i<argc; i++)
        printf("Hello, %s\n", argv[i]);
.
wq
EOS
```

```
83
142
```

```
git add hello.c
git commit -m "Greets all the people passed."
```

```
[main 12a40a6] Greets all the people passed.
1 file changed, 3 insertions(+), 2 deletions(-)
```

```
make hello
./hello Alice Bob
```

```
cc    hello.c -o hello
Hello, ./hello
Hello, Alice
Hello, Bob
```

One more edit...

```
ed hello.c <<EOS
4s/0/1/
wq
EOS
git add hello.c
git commit -m "Stops greeting the program itself."
```

```
142
142
[main 8147cde] Stops greeting the program itself.
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
make hello
./hello Alice Bob
```

```
cc hello.c -o hello
Hello, Alice
Hello, Bob
```

So what have we done?

So far we've made three changes to our code: lets see what these look like in Git!

```
git log --oneline | cat
```

```
8147cde Stops greeting the program itself.  
12a40a6 Greets all the people passed.  
6f436f6 Initial commit of the greeting program.
```

Tags, branches and HEAD...

Commits are all *identified* by their hash...

- ▶ but you can name specific commits by using the `git tag` command
- ▶ (this is useful for marking releases or submitted versions of your code)

All commits are made to a *branch* which is a *tag*

- ▶ When the commit is made the *branch* tag is *updated* to point to the new commit at the top of *branch*.
- ▶ The *default* branch is usually called `main` (or `master`)
- ▶ (This is wrong in all important respects; but it's an okay simplification)

There is also a *special* tag called HEAD

- ▶ Always points to wherever your code is currently at
- ▶ Minus any unstaged work

Working with commits

Say you've made a bunch of changes to a file, but not committed them. You'd like to throw away the changes you made:

```
git checkout HEAD -- hello.c
```

Or if you've changed a lot of stuff and want to go back to clean:

```
git reset --hard HEAD # Remove all changes
git clean -dfx # Delete all untracked files
```

```
HEAD is now at 8147cde Stops greeting the pro...
Removing hello
```

Say you'd like to go back to how the code was *before* the last commit:

```
git checkout HEAD~1
```

Say you're done looking at the code in an old state, and want to go back to working on the main branch:

```
git checkout main
```

Say a commit was a horrible mistake and you'd like to apply it in reverse and undo all the changes of it:

```
git revert HEAD
```

```
[main de70029] Revert "Stops_greeting_the_pro...
Date: Thu Jan 25 16:33:42 2024 +0000
1 file changed, 1 insertion(+), 1 deletion(-)
```


So far we've just been working on our own code

We've just been tracking local changes...

- ▶ But Git was made to let people share code

Lets make it more complex!

Rather than making *our own* new repo:

- ▶ Let's take a copy *or clone* someone else's
- ▶ And let's share those changes with them

What do we mean by *decentralized*?

We said Git was a *decentralized version control system*

- ▶ As opposed to a *centralized* one like SVN/CVS

What this means in practice is that your local repo *should* have the complete history of the repo.

- ▶ And should be able to function as a master copy of the repo.
- ▶ (Again this is a simplification but go with it...)

Let's pretend Alice has a Git repo of their course groupwork in `~alice/coursework/`

- ▶ Bob wants to collaborate with Alice and get their own copy

So Bob runs...

```
git clone ~alice/coursework ~bob/coursework
```

```
Cloning into '~bob/coursework'...  
done.
```

```
cd ~bob/coursework  
git log --oneline
```

```
af3818c States true facts about this course  
7eb311c First draft of the coursework
```

```
make coursework
```

```
cc -O2 -pipe -o coursework coursework.c
```

```
./coursework
```

```
Softwaer tools is cool!  
Hello World!
```

Oh no: a mistake!

Bob can fix that for Alice!

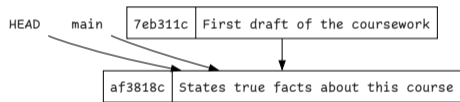
```
$ ed coursework.c
124
4c
    printf("Software_tools_is_cool!\n");
.
wq
124

$ git add coursework.c

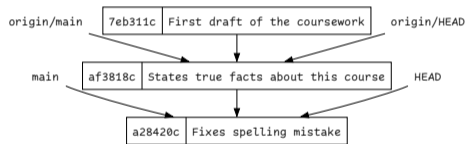
$ git commit -m "Fixes spelling mistake"
[main a28420c] Fixes spelling mistake
1 file changed, 1 insertion(+), 1 deletion(-)
```

So now

On Alice's repo



On Bob's repo



So how do we get Bob's changes back to Alice?

There are multiple ways!

Bob can send Alice their changes patch based

Alice can pull Bob's changes pull based

Patch based approach (Bob's end)

This is how the *Linux Kernel* and many other open source projects manage commits.

Bob starts by preparing a *patch*

```
$ git format-patch origin/main \  
  --to=alice@bristol.ac.uk \  
0001-Fixes-spelling-mistake.patch
```

And sends it to Alice

- ▶ (check out =git send-email=!)
 - ▶ (You'll probably need to configure sendmail for that...)

```
From a28420cd5c45d06c9a51625d5a03c37bb77e2ca9 Mon Sep 17 00:00:00 2001  
From: bob <bob@bristol.ac.uk>  
Date: Tue, 22 Nov 2022 11:53:04 +0000  
Subject: [PATCH] Fixes spelling mistake  
To: alice@bristol.ac.uk  
  
---  
coursework.c | 2 +-  
1 file changed, 1 insertion(+), 1 deletion(-)  
  
diff --git a/coursework.c b/coursework.c  
index 2e191f8..8927b2f 100644  
--- a/coursework.c  
+++ b/coursework.c  
@@ -1,7 +1,7 @@  
#include <stdio.h>  
  
int main(void) {  
- printf("Softwaer tools is cool!\n");  
+ printf("Software tools is cool!\n");  
  printf("Hello World!\n");  
  return 0;  
}  
--  
2.38.1
```

Patch based approach (Alice's end)

Alice reviews Bob's patch and, if they like it... applies it to their tree.

```
$ git am ../bob/0001-Fixes-spelling-mistake.patch
Applying: Fixes spelling mistake

$ git log --oneline
575dcde Fixes spelling mistake
af3818c States true facts about this course
7eb311c First draft of the coursework
```

The ID is different however to Bob's tree for the latest commit

- ▶ (because *Alice* committed it).

If Bob wants to keep IDs in sync with Alice they need to re-clone

- ▶ (or `git fetch origin`).

Aside

Technically `git am` is actually a couple of git commands rolled into one...

- ▶ First it runs `git apply` with the patch to stage all the changes it will make
- ▶ Then it runs `git commit` with the commit message also supplied in the patch

There are a lot of git commands that are really lower level commands chained together—watch out for them!

The alternative

Some people **hate** the patch workflow.

- ▶ Who configures sendmail nowadays?

The alternative is to let Git do the work for you and trust the other person.

Bob tells Alice they fixed a mistake.

- ▶ Alice adds Bob's repo as a remote

```
$ git remote add bob ~bob/coursework  
  
$ git fetch  
remote: Enumerating objects: 5, done.  
remote: Counting objects: 100% (5/5), done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), 255 bytes | 127.00 KiB/s, done.  
From ~bob/coursework  
* [new branch] main      -> bob/main
```

Alice inspects Bob's changes...

File Edit View Help

<ul style="list-style-type: none">remotes/bob/main Fixes spelling mistakemain States true facts about this courseFirst draft of the coursework	bob <bob@bristol.ac.uk> alice <alice@bristol.ac.uk> alice <alice@bristol.ac.uk>	2022-11-22 11:53:04 2022-11-22 11:15:29 2022-11-22 11:07:56
--	---	---

SHA1 ID: a28420cd5c45d06c9a51625d5a03c37bb77e2ca9 ← → Row 1 / 3

Find commit containing: Exact All fields

Search

◆ Diff ◆ Old version ◆ New version Lines of context: 3 Ignore space change Line diff

◆ Patch ◆ Tree


Comments
coursework.c

Author: bob <bob@bristol.ac.uk> 2022-11-22 11:53:04
Committer: bob <bob@bristol.ac.uk> 2022-11-22 11:53:04
Parent: [af3818ce392c983a2d5523ef7b43f5e294bd674e](#) (States true facts about this course)
Branch: [remotes/bob/main](#)
Follows:
Precedes:

Fixes spelling mistake

----- coursework.c -----
index 2e191f8..8927b2f 100644
@@ -1,7 +1,7 @@
#include <stdio.h>

```
int main(void) {  
- printf("Softwaer tools is cool!\n");  
+ printf("Software tools is cool!\n");  
}
```



And if they're happy...

```
$ git pull bob main
From ~bob/coursework
 * branch      main      -> FETCH_HEAD
Updating af3818c..a28420c
Fast-forward
 coursework.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git log | cat
commit a28420cd5c45d06c9a51625d5a03c37bb77e2ca9
Author: bob <bob@bristol.ac.uk>
Date: Tue Nov 22 11:53:04 2022 +0000

    Fixes spelling mistake

commit af3818ce392c983a2d5523ef7b43f5e294bd674e
Author: alice <alice@bristol.ac.uk>
Date: Tue Nov 22 11:15:29 2022 +0000

    States true facts about this course
```

...

Aside

Again the `git pull` command is really a composite. The following is equivalent:

```
$ git fetch bob

$ git merge --ff bob/main
Updating af3818c..a28420c
Fast-forward
 coursework.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Locality matters

So far we're dealing with files on a local filesystem.

- ▶ We cloned from a direct path to another user's Git repo
- ▶ Git *in-the-plumbing* is just a protocol for how you move files about and what you stick in that `.git` folder and how you manipulate them.

We've been accessing those files via a local shell:

- ▶ But Git doesn't care
- ▶ HTTP has a filesystem API
- ▶ SSH lets you access remote shells

Wouldn't it be *neat* if instead of having to do everything locally we could have a *centralised forge* where we go and get all our changes and send them back when we're done?

Github is **not** git.

Github gives you a *centralised* remote (called a *forge*, and usually some build automation and issue tracking software):

- ▶ You can sign up for an account
- ▶ Set up access for users
- ▶ And then centrally send commits to everyone with the `git push` command
- ▶ Can host a project page and build infrastructure too

Github is owned by Microsoft:

- ▶ Some people don't like that
- ▶ Some ~~questionable~~ *naughty* behaviour surrounding AI and Open Source

Alternatives:

<https://bitbucket.org> Owned by Atlassian

<https://gitlab.com> Can self-host

<https://sr.ht> Owned by Drew DeVault... costs money (but it's really good)

Self host? All you need is a server (search for *bare repositories* to find out how)

To use a forge

In Bob's repo

```
$ git remote set-url origin \  
  git@github.com:alice/coursework  
  
$ git status  
On branch main  
Your branch is ahead of 'origin/main' by 1 co...  
  (use "git push" to publish your local commits)  
  
nothing to commit, working tree clean  
  
$ git push  
Everything up-to-date
```

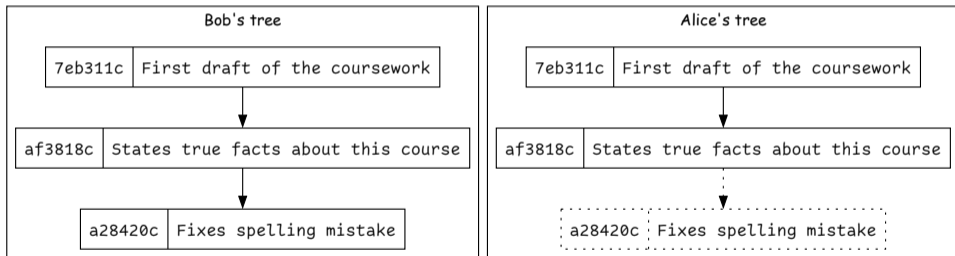
In Alice's repo

```
$ git remote -v  
remote git@github.com:alice/coursework (fetch)  
remote git@github.com:alice/coursework (push)  
  
$ git pull remote main  
From git@github.com:alice/coursework  
* branch      main      -> FETCH_HEAD  
Updating af3818c..a28420c  
Fast-forward  
  coursework.c | 2 +-  
  1 file changed, 1 insertion(+), 1 deletion(-)
```

So what happens when things go wrong?

When Alice pulled Bob's changes earlier they could be *fast-forwarded*.

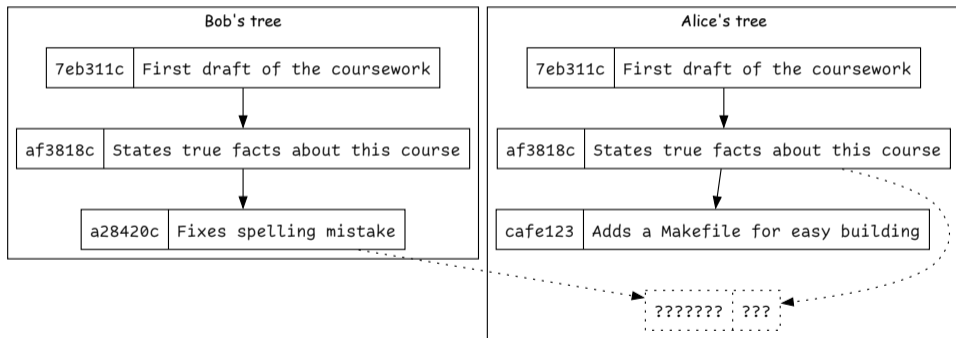
- ▶ This means that the changes could be pulled straight across and copied into Alice's tree.



Busy, busy, busy...

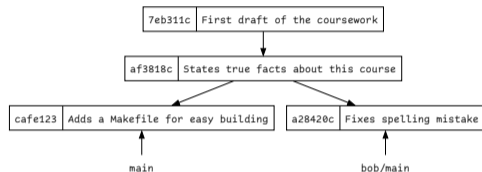
What about if Alice has been busy and made some commits of their own.

- ▶ The fast forward can't happen now because the trees have *diverged*



Merging

From Alice's point of view this is what the trees look like

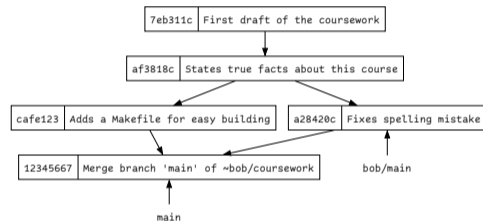


```
git merge --no-ff bob/main
```

```
hint: Waiting for your editor to close the file.  
Merge made by the 'ort' strategy.
```

```
Makefile | 0  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 Makefile
```

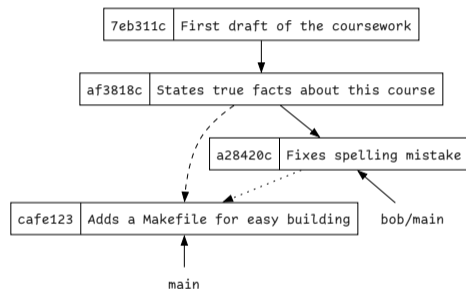
The simplest approach is to do a *merge* and add a commit explicitly merging the changes from both paths of the tree



(But normally it'll be smart and spot that you changed different files and still do the *fast-forward...*)

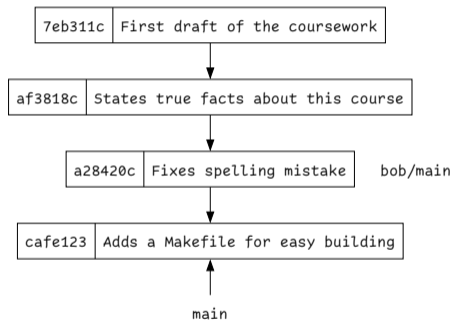
Rebasing

Alternatively Alice could do a bit of time-travelling... Lets *pretend* that Alice's commit came after Bob's:



```
$ git rebase bob/main  
Successfully rebased and updated refs/heads/main.
```

Then we can *fast-forward* as before through Bob's change and then replay Alice's new commit after.



Now we get a nice neat straight line tree again!

Merging vs Rebasing

Merging is simpler *conceptually*...

- ▶ ...but messy

Rebasing is neater

- ▶ ...but complicated and prone to failure
- ▶ There are some other neat tricks you can do that make it much better (e.g. `git bisect`)

You (and your employers) will have an opinion

- ▶ Do that.
- ▶ It really doesn't matter
- ▶ (I slightly prefer the merge version but I switch back and forth...)

So far easy!

So far our merges have been *easy*.

- ▶ Alice and Bob have made edits to different files
- ▶ Changes have all been able to be done by Git automatically.
What happens if Alice and Bob *both change the same lines in the same file?*

```
printf("Hello World");
```

Alice → printf("HeLo Byd");

Bob → printf("Dydh da");

```
$ git merge bob/main
Auto-merging coursework.c
CONFLICT (content): Merge conflict in coursework.c
Automatic merge failed; fix conflicts and then commit the result.
```

Lets fix the conflict

Git has discovered there are two sets of changes and it can't work out which is the one to go with...

If we *follow Git's instructions* coursework.c looks like:

```
#include <stdio.h>

int main(void) {
<<<<<<< HEAD
    printf("Helo Byd\n");
=====
    printf("Dydh da\n");
>>>>>>> bob/main
    return 0;
}
```

Fix up the file and then run `git add / git commit` when it looks good...

- ▶ Don't just delete one side of it.
- ▶ ...Seriously... I've seen people fired for that.

```
$ git add coursework
$ git commit
[main 16d3aa6] Merge remote-tracking branch 'bob/main'
```

Wrap up of remotes

- ▶ Use `git remote` and `git clone` to work with other people
- ▶ Use `git fetch` or `git pull` or `patch` files to get other peoples work
- ▶ Use `git merge` or `git rebase` to integrate changes
- ▶ Use `git push` to send work back to a forge
- ▶ Merge conflicts are a pain but you have to deal with them

Merge tools

If you find yourself dealing with merge conflicts regularly... there are tools that help you work with them

<https://meldmerge.org> Good tool for dealing with merges

(I use Emacs.)

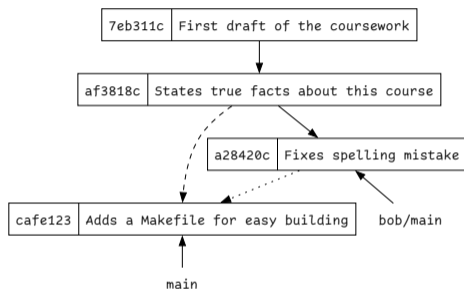
Forking timelines

So far we've had a local copy of the code and a remote one...

- ▶ But why should Bob have all the fun?
- ▶ What if Alice wants to have her own versions of the code with separate sets of changes?

Lets talk about *branches* and show some tricks for working with them!

Last time...



We had this situation where *Alice* and *Bob*'s trees had diverged...

- ▶ ...but they had a shared history
- ▶ ...and we could bring them back together

But why do we need to restrict this to just other people's trees?

Branch refresher

A branch is a tag to the last commit in a trail of commits that gets updated every time you make a new commit to that branch.

Branching

The plan

Lets aim to keep the main branch clean

- ▶ The main branch always works

When we do some work we take a branch off of main

- ▶ (or possibly some other sensible place)
- ▶ Do the work...
- ▶ Merge back in when done.

Lets give it a go!

```
git branch new-feature main  
git checkout new-feature
```

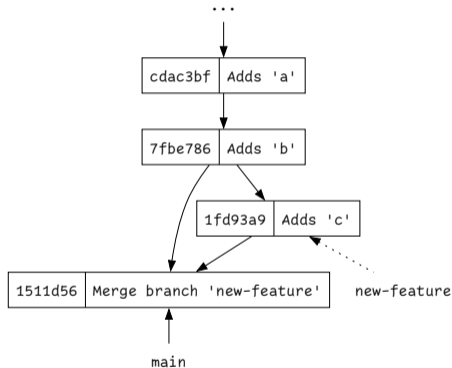
```
Switched to branch 'new-feature'
```

```
touch c; git add c; git commit -m 'Adds c'
```

```
[new-feature 1fd93a9] Adds c  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 c
```

```
git checkout main; git merge new-feature
```

```
Switched to branch 'main'  
Merge made by the 'ort' strategy.  
c | 0  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 c
```



Why on earth would you do this?

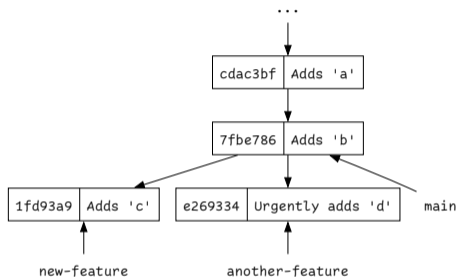
It starts to come in handy when you're working on *multiple features* at once.

Say whilst you're developing your new-feature, your friend needs you to urgently work on another-feature...

- ▶ You don't want to merge new-feature in yet though because you're still working on it.
- ▶ You don't want to add unrelated code for a new-feature in with the work for another-feature.

```
$ git branch another-feature 7fbe786
$ git checkout another-feature
Switched to branch 'another-feature'

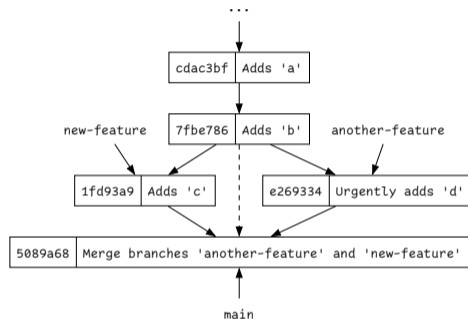
$ touch d
$ git add d; git commit -m 'Urgently add d'
[another-feature e269334] Urgently add d
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644
```



Merge them all!

```
$ git checkout main
Switched to branch 'main'

$ git merge --no-ff another-feature new-feature main
Merge made by the 'octopus' strategy.
 c | 0
 d | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c
 create mode 100644 d
```



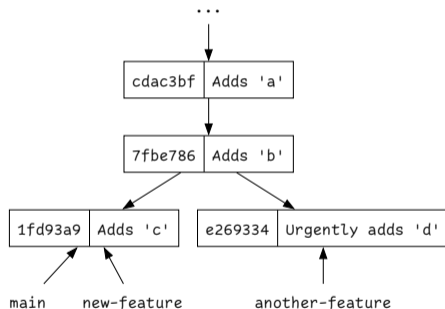
Normally you wouldn't bother with the `--no-ff` and the dashed line would disappear!

- ▶ Sometime you like the extra info (especially when teaching)
- ▶ But normally it's just noise...

Well except...

Normally you *wouldn't* do this

- ▶ What if merging another-feature breaks something in new-feature?
- ▶ It would be nice to test things before merging!
- ▶ But new-feature doesn't have the work now merged into main!



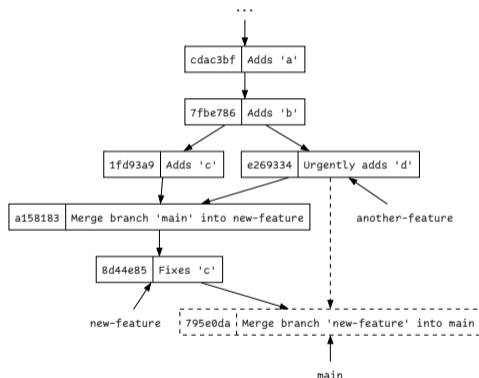
We could merge main into new-feature

We could try merging main into new-feature...

- ▶ Test and see if any extra changes are needed...
- ▶ Add extra commits as required
- ▶ Then merge the new-feature *back into* main

Still a bit messy as we're going to get *at least one* merge

- ▶ (assuming we don't disable fast-forwarding)



Instead we could rebase

What I'd normally do is *rebase* new feature on main

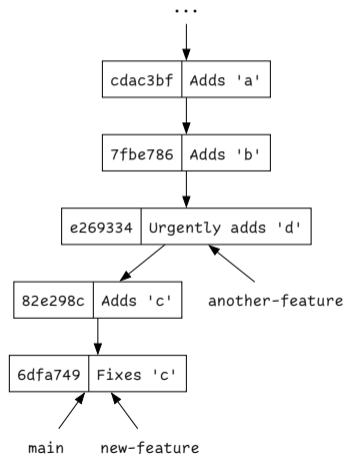
- ▶ Essentially rewrite history so it looks like new-feature was done after the merge of another-feature
- ▶ Fix any conflicts then and there as part of the original Adds 'C' commit
 - ▶ (potentially changing its ID)
- ▶ **Then** re-test and fix issues and commit
- ▶ Then merge back into main.

```
$ git rebase main new-feature  
Successfully rebased and updated refs/heads/new-feature.
```

```
$ git checkout new-feature  
Already on 'new-feature'
```

(I always get the rebase command the wrong way round)

- ▶ (Seriously, it took 3 attempts...)
- ▶ (Always make a backup before rebasing)



We can do more with rebase!

Suppose we were going to send new-feature as a series of patches to merge by a project maintainer

- ▶ `git format-patch` would generate one patch for each commit
- ▶ And that's fiddly for the maintainer to apply
- ▶ And it'd mean that we have commits where the whole thing is broken before we fixed C.

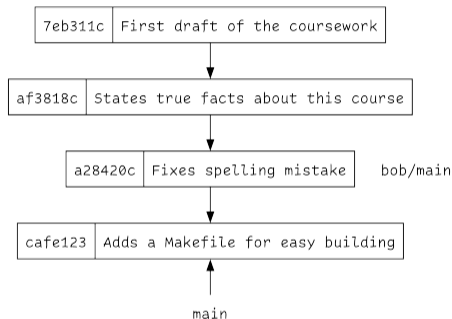
Rebase lets us edit the repository history!

So once we've done the fix we can rebase a branch interactively and decide what to do

```
$ git rebase -i main new-feature
[detached HEAD 7d2180a] Adds c, and fixes it..
Date: Fri Nov 25 14:31:08 2022 +0000
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 c
[detached HEAD 5af45b8] Adds c, and fixes it..
Date: Fri Nov 25 14:31:08 2022 +0000
1 file changed, 1 insertion(+)
create mode 100644 c
Successfully rebased and updated
refs/heads/new-feature.
```

This is considered a professional courtesy amongst software engineers

- ▶ Also good for hiding all those *argh I broke it* commits
- ▶ And removing swearing before you send it to the customer



Warning!

Being too clever with rebase will break your repo

Sometimes in unfixable ways

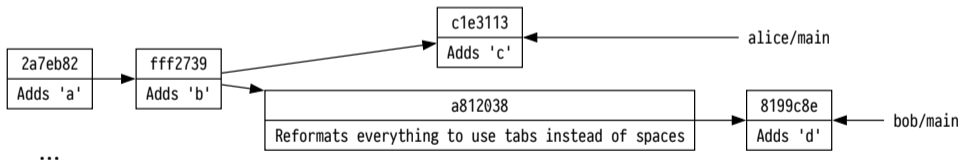
- ▶ *Always* backup before being clever
- ▶ rebase is considered *advanced* Git

One last trick...

Suppose Bob has done some interesting work on their main branch, and also some less interesting work,.

- ▶ They've fixed some bugs,
- ▶ But they've also switched all your files from using spaces to tabs

How do you cherry-pick the things you want and ignore the things you don't?



git cherry-pick

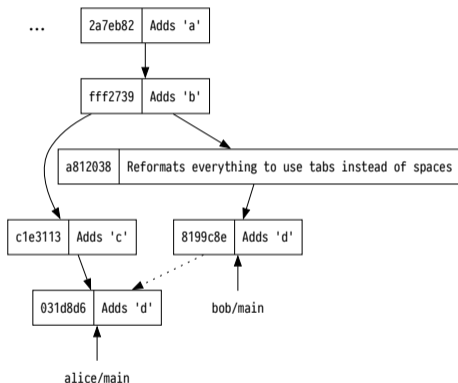
```
$ git cherry-pick 8199c8e
[main 031d8d6] Adds 'd'
Date: Mon Nov 28 09:10:43 2022 +0000
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 d
```

Why would you do this?

- ▶ Git will generally be *somewhat* smart about how it copies the work over
- ▶ If it needs more commits it'll pull them too
- ▶ If you merge later, Git will be *somewhat* smart about where things came from and *maybe* not cause a conflict

Internally this is just a rebase...

- ▶ But Git hides a lot of the complexity
- ▶ ...actually everything in Git is really just a *rebase* with a nicer UI ; -)



Wrap up

Right that's Git!

- ▶ There are infinitely more things you can do with it
- ▶ ...but *hopefully* this is 90% of what you'll *normally* do

Golden rules

- ▶ Do not break the build
- ▶ Write helpful log messages
- ▶ Rebase with fear (but you do have to do it sometimes)

DO NOT BREAK THE BUILD

- ▶ If you did it in the more professional places I worked; you stayed late til it was fixed
- ▶ If you did it in a startup I worked; you stayed late til it was fixed and you owed everyone a beer and got called names
- ▶ If you do it in IBM or Google; you were fired.
 - ▶ At least according to my old Prof (Awais Rashid)