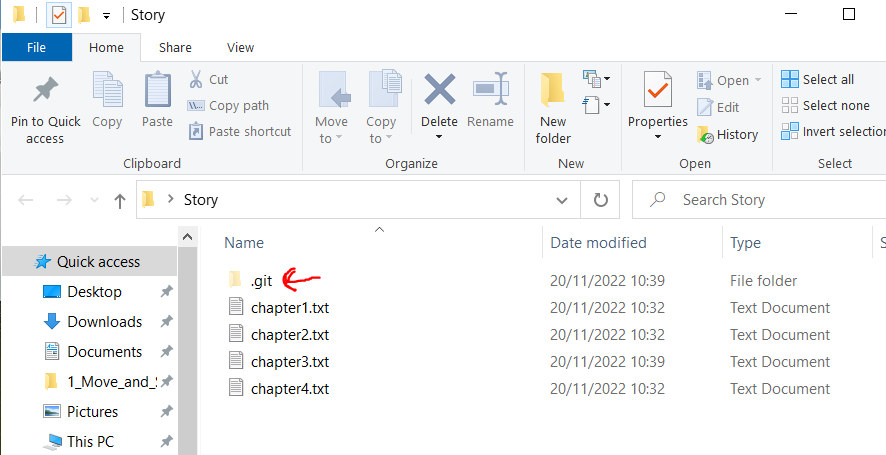
**Local Repository, Command Line and Version Control**

The command line is used to implement git locally (i.e. saving our files locally, on our computer)

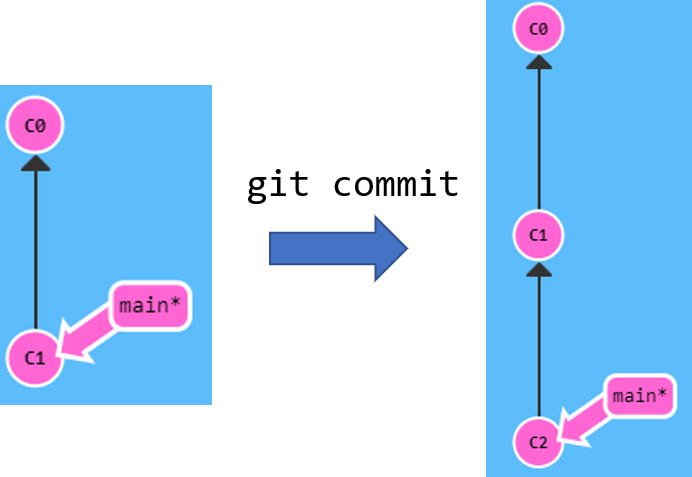
* **git init:** initializes an empty git repository inside the working directory.
* **git status**: see what is currently inside your staging area. Untacked files or files that have been changed since the last commit are shown in red. Tracked files are shown as green, and ready to be committed.
* **git add *<file name>*:** Adds file to the staging area.
* **git add .:** Adds every file in the working directory to the staging area.
* **git rm --cached -r *<file name>:*** Removes file from staging area
* **git rm --cached -r .*:*** Removes all files from staging area
* **git commit -m “*commit message”*:** Commits a specific file, i.e. creates a save point. The “commit message” should describe the changes that were made. By convention, the commit message should be written in the present tense.
* **git log:** You can see what commits you have made (time, author, id and commit message) in the working directory.
* **git diff *<file name>***: shows what was modified in a specific file. It will show in red what was removed, and in green what was added.
* **git checkout *<file name>:*** roles back a specific file to the last version that was committed in our local repository.
* **git restore *<file name>*:** discards all the changes done in the working directory (uncommitted changes)
* **git clone <repository link>:**

When we do **git init**, it creates a local repository in our working directory called ***.git***.

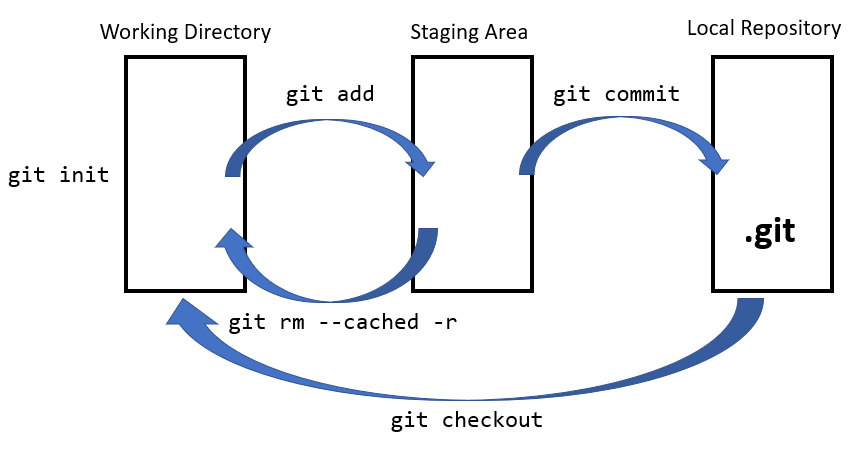


**Staging area:** an intermediate place where you can pick and choose which files inside your working directory you want to commit.

**Commit:** Similar to saving a file that's been edited, a commit records changes to one or more files in your branch. Git assigns each commit a unique **ID**, called a SHA or **hash**, that identifies the **specific changes** and **when the changes were made**.



The image below shows the summary of how the workflow works while working in the local environment, which is what we have learned so far.



**Pushing your local repository into a remote repository**

git remote add origin https://github.com/Filipeesf19/Web-Development.git

git branch -M main

git add .

git commit -m “Bug Fixed”

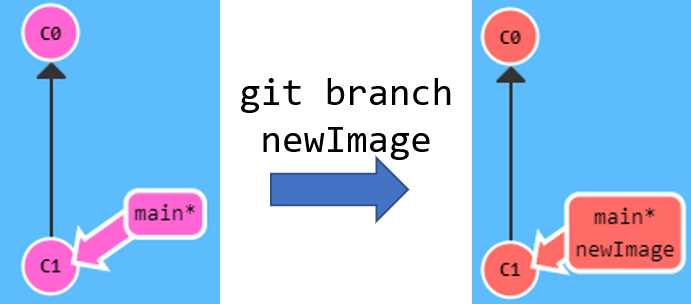
git push -u origin main

**Pulling your remote repository into you repository**

git clone https://github.com/Filipeesf19/Web-Development.git

**Branching**

In Git, branches are a part of your everyday development process. Git branches are effectively **a pointer to a snapshot of your changes**. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes.

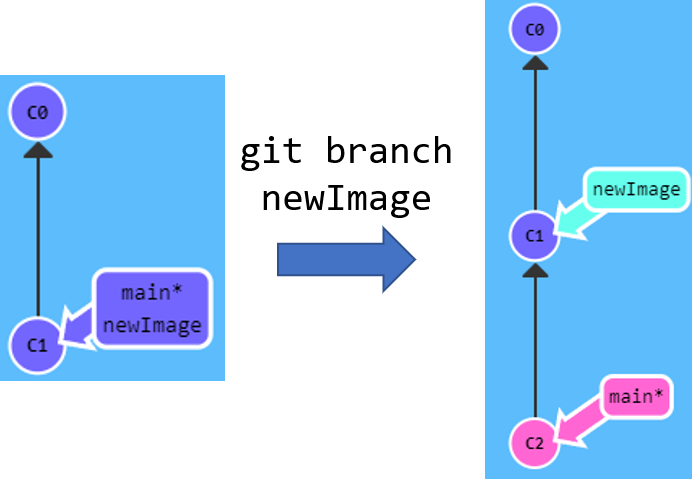


Branching is needed because sometimes you are developing new features, sometimes you are fixing bugs and all of these things may break your main project so you don't want to do it on the master branch.

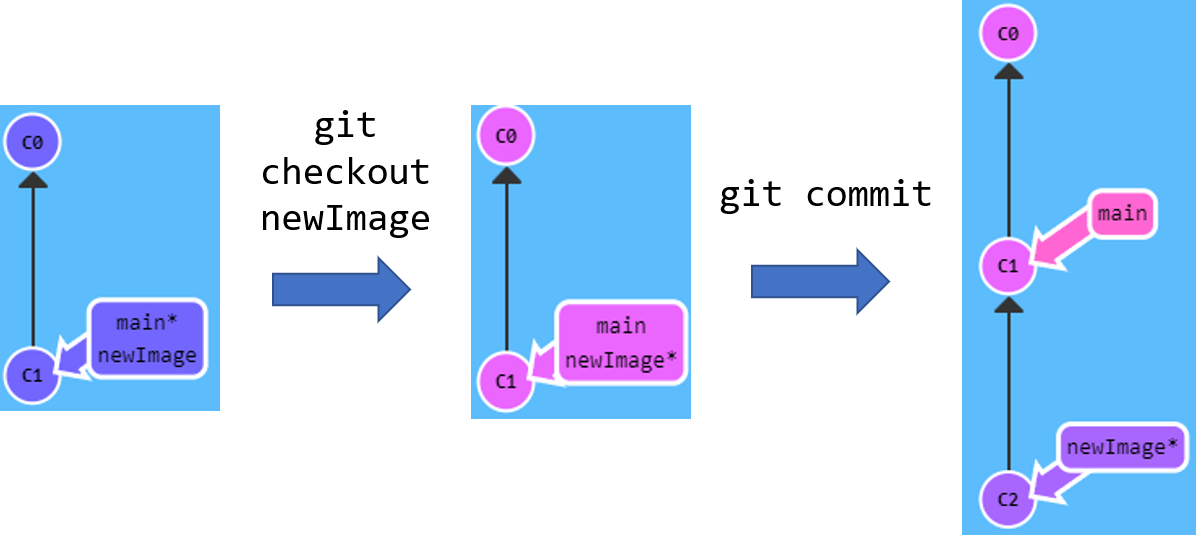
You only want to put it onto the master branch once you've tested, once you know that everything is working fine and then you can bring your code back to the working copy ready for shipment and deployment.



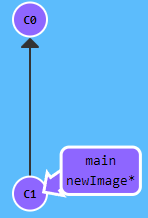
Note that to move the branch, this has to be targeted. For example, in the image below, the **main** branch moves but the **newImage** didn’t. That's because we weren't "on" the new branch, which is why the asterisk (**\***) was on main.



Doing git checkout **newImage** will put us on the new branch before committing our changes, and our changes will be recorded on the new branch.



One way to do create a new branch and check it out straight away is **git checkout -b newImage**.

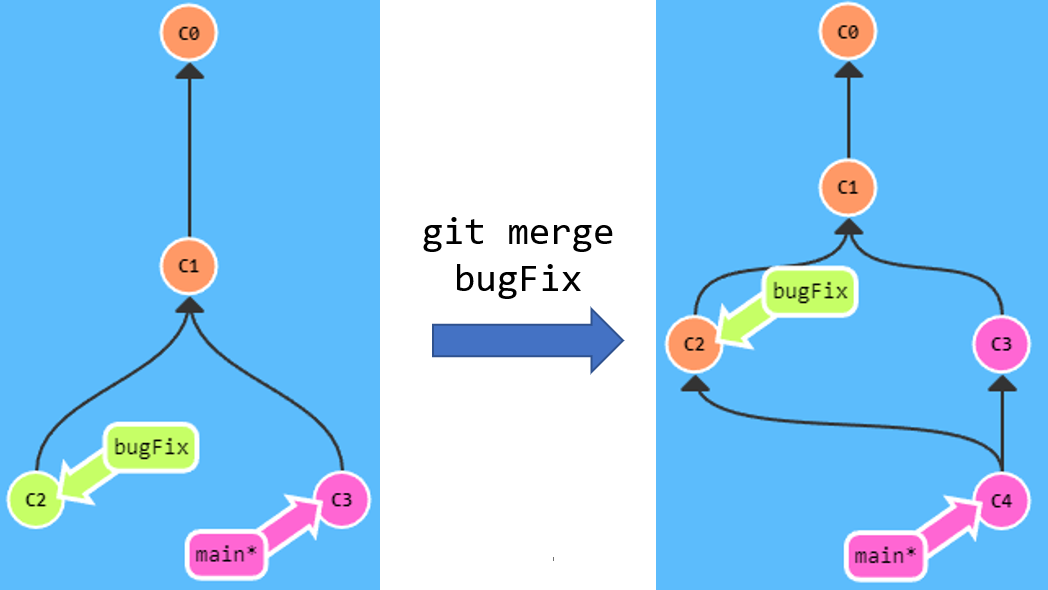


**Merging**

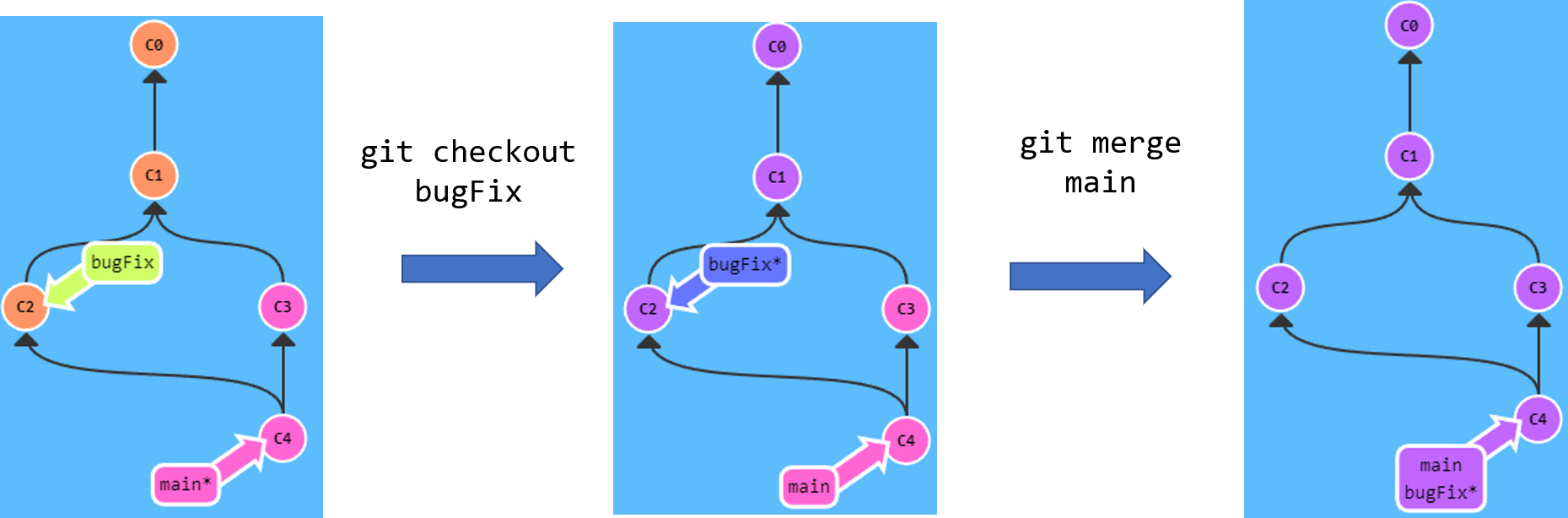
Now we need to learn on how to combine the branches to the main branch.

One of the methods is **git merge**. Merging creates a new commit with two parents:

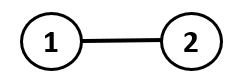
* one contains all the work done in the main branch
* one contains all the work done in the new branch



The **main** branch is now blended into all the commits, but the **bugFix** is not. To do that, we have to check it out and merge it into the last commit. Now each branch contains all the work in the repository.

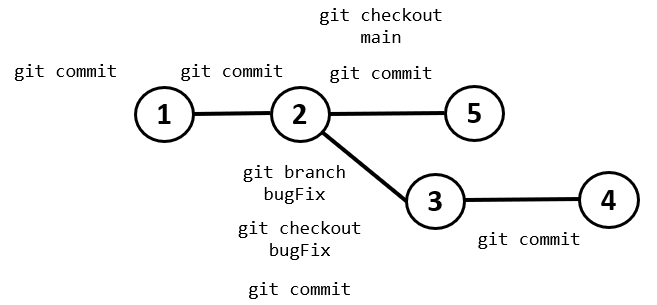


Now let's start off with a simple example say if we had version 1 and 2, so two commits that were made to our local repository.



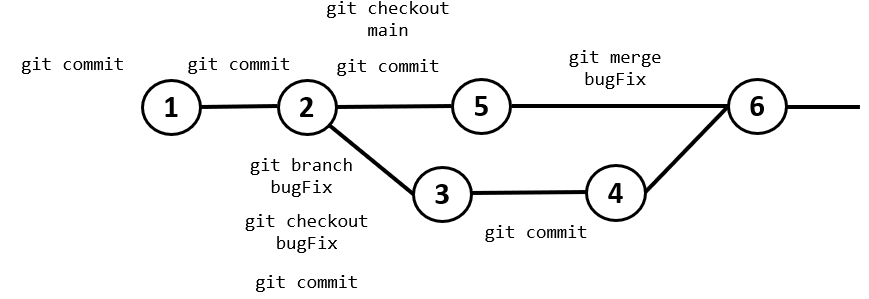
After the second commit, we **create a new branch** and we start committing to this new branch or this experimental branch.

Now simultaneously **we can continue working on the main branch** putting out all those essential updates or bits of code that are maintaining our main project, but at the same time we can continue to update and work on this experimental branch trying things out and committing our experiments to this experimental branch. So now we have **two branches that are parallel** to each other and **they can be developed simultaneously**.



If at some point in the future that we decide that that experiment was really fruitful and the feature that we built in it was really great and we'd like to merge it back to the main branch or to the main project, then that can be done really easily as well by simply placing a **merge request** in.

And we can bring all of those changes that we experimented with, that we messed around with back to the main project and check to see if there's any conflicts with the main branch code.



**Git Rebase**

The second way of combining work between branches is **rebasing**. This **moves the entire feature branch** **to begin on the tip of the main branch**, effectively incorporating all of the new commits in “***main***”.

The advantage of rebasing is that it can be used to make a **nice linear sequence of commits**. The commit log / history of the repository will be a lot cleaner if only rebasing is allowed.

|  |  |  |
| --- | --- | --- |
| **Code** | **Description** | **Image** |
| - | Initial code with 2 commits and just one main branch |  |
| git branch bugFix | Created the bugFix branch |  |
| git checkout bugFix | Checked out bugFix so we are now targeting it |  |
| git commit | Did the changes to bugFix and committed it |  |
| git checkout main | Checked out main so we are now targeting it |  |
| git commit | Did the changes to bugFix and committed it (we now have to branches) |  |
| git checkout bugFix | Checked out bugFix so we are now targeting it |  |
| git rebase main | Moves bugfix to the main branch as the last commit |  |

**GitHub and Remote**

You can transfer your local repository to a server (GitHub), making it a **remote repository**. To do that, go to your working directory of your project using Hyper and insert these three lines of code:

**git remote add origin** [**https://github.com/Filipeesf19/Story.git**](https://github.com/Filipeesf19/Story.git)

**git branch -M main**

**git push -u origin main**

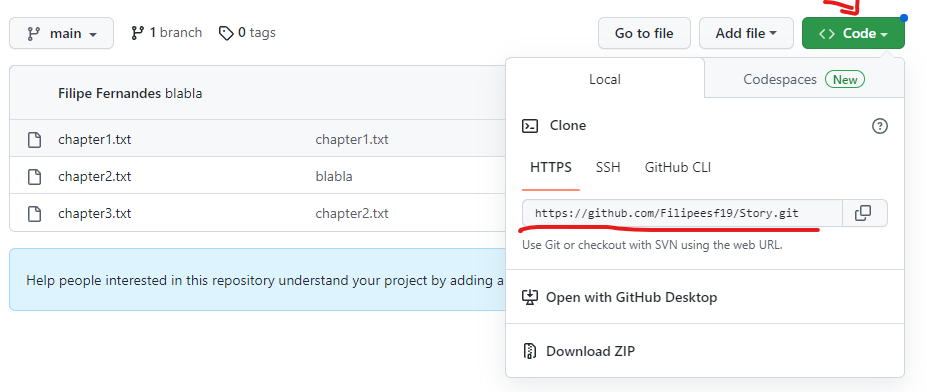
The first line tells the local git repository that we have **created remote repository** on the internet. The link there is simply the link of your repository, which can be obtained in your GitHub repository (see image below). ***Origin*** is simply the name of your remote repository, and it can have any name. By convention, keep the name ***origin***.

The naming convention for a remote branch is then as follows:

**<remote name>/<branch name>**

For example:

**origin/bugFix**

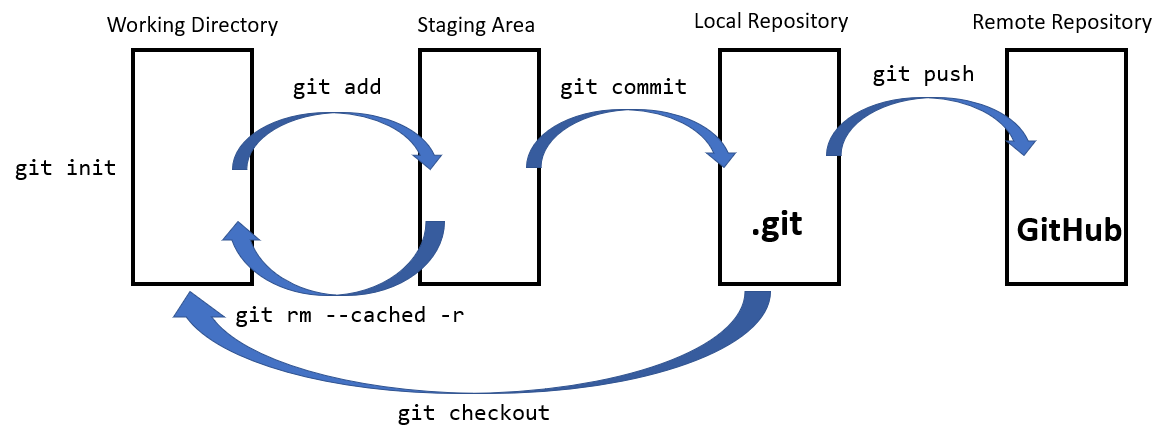


The third line of code **pushes** the local repository to our remote depository, using the flag ***u***, which links the local repository to the remote repository. The name of the remote repository is origin, the name of the branch is main (more on branches later).

Remote repositories are just copies of your repository on another computer, which you talk through the Internet. This allows you to transfer commits back and forth.

The two main advantages of using remote repositories are:

* Back up your work – it allows you the restore files to a previous state.
* **Work with a team** – it makes it easier to work with other people on the same projects.

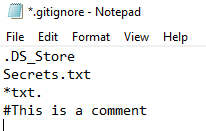


**Gitignore**

You might not want to publish things like passwords of API keys on an open platform like GitHub.

To do that, we need to create a ***gitignore.txt*** file inside our working directory and write there anything we want to not go to the staging area.

Note: in the .gitignore.txt file, you can use ***#*** to add comments, or ***\**** if you want certain extension to be ignored. For example:



**Cloning**

Cloning is the act of **pulling the commits from the remote repository to your local machine** and **store the files inside your working directory**.

To do that, you just have to type git clone in Hyper, plus the link of the repository, which is found on the top-right corner of your GitHub repository (see GitHub and Remote lesson):

**git clone** [**https://github.com/Filipeesf19/Story.git**](https://github.com/Filipeesf19/Story.git)

That means we have to go though all the steps again to update the Remote Repository by:

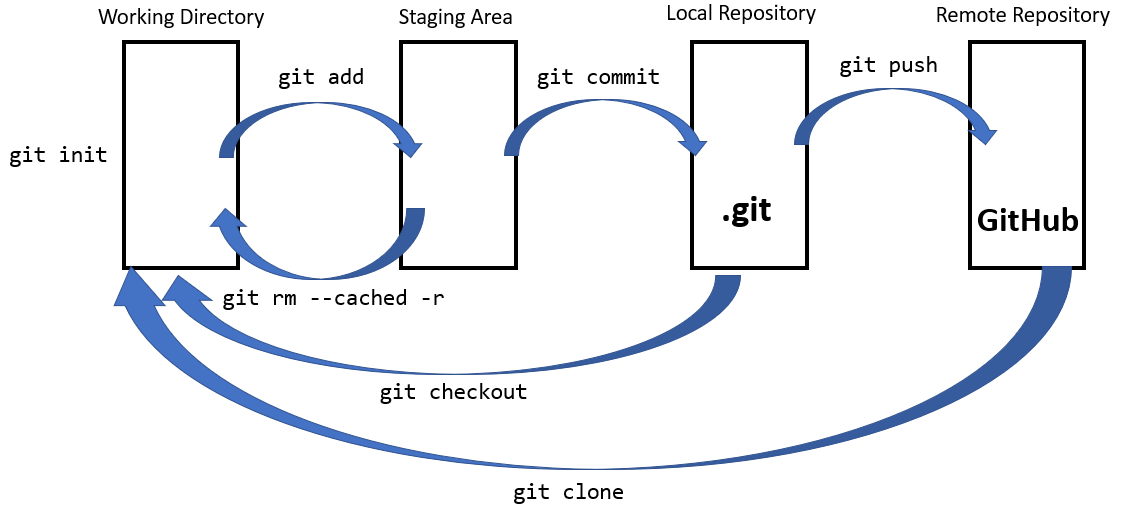
1. Doing the changes to the project

2. Doing **git add .** to add all the files to the stating area

3. doing **git commit -m “commit message”** change it to the local repository ***.git***.

4. Doing:

* **git remote add origin** [**https://github.com/Filipeesf19/Story.git**](https://github.com/Filipeesf19/Story.git)
* **git push -u origin main**



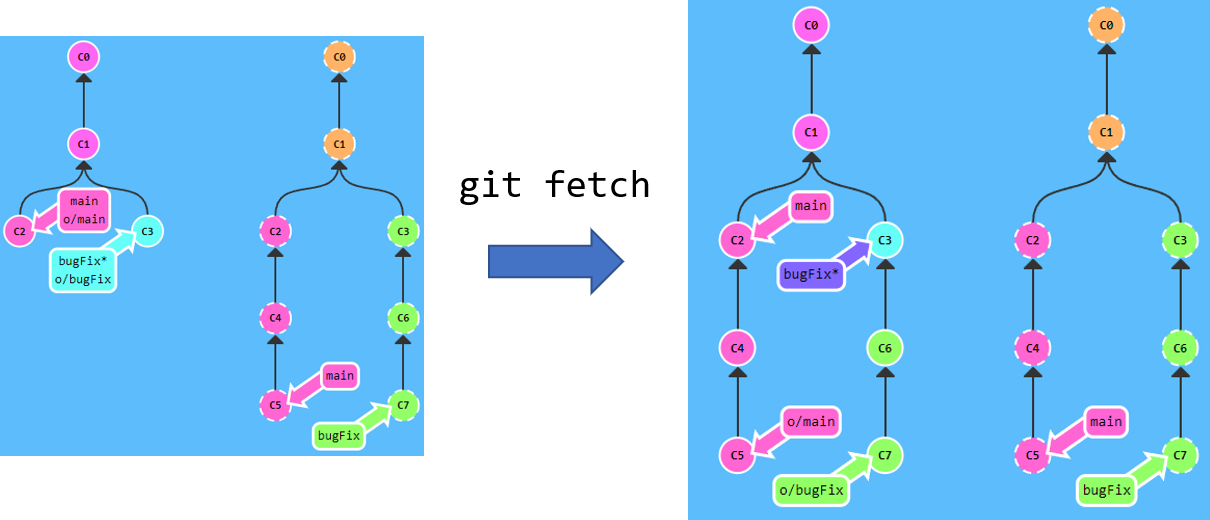
**Git Fetch**

**git fetch** synchronizes the local repository with the remote repository, performing two main steps:

* Downloads the commits that that the remote repository has but are missing in the local repository.
* Updates where our remote branches point (for instance, **origin/main**).

**git fetch** doesn’t change anything about your local state. It will not update your **main** branch or change anything about how your file system looks right know. I will only download the necessary date to do that, but **it doesn’t not actually change your local files**. You can think of it as just one step of downloading.

In the example below, **git fetch** brought all the commit missing in the local repository from the remote repository (**C4**, **C5**, **C6** and **C7**). So, the remote branches **origin/main** and **origin/bugFix** are now up to date comparing to our remote repository. However, the local branches **main** and **bugFix** are still on commits **C2** and **C3**.



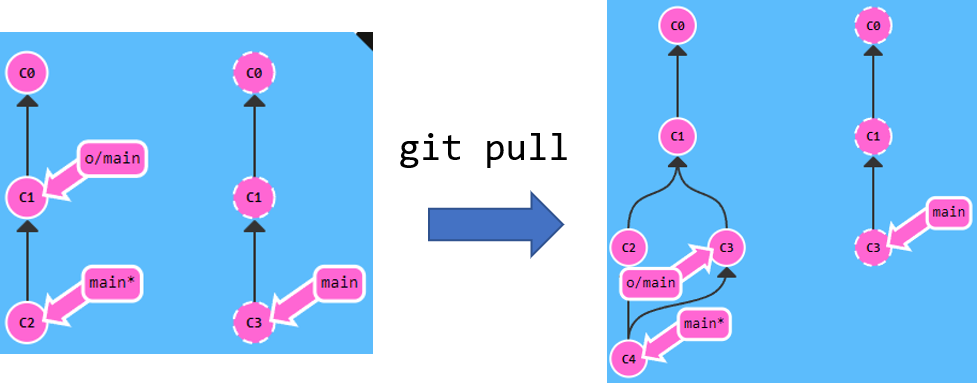
**Git Pull**

**git pull** is nothing else than a combination of **git fetch** and **git merge**.

For example, see the example below, where we have a remote repository with commits **C0**, **C1** and **C3**. We want to update our local repository so it reflects what we’ve got in the remote one.

So what **git pull** does is:

1. it creates a new branch in the last commit **C1**.
2. Get’s hold of the commit that we don’t have locally (**C3**)
3. Merges the new branch with the local branch, creating a new commit **C4**, which is essentially a merged version of **C3** and **C2**.

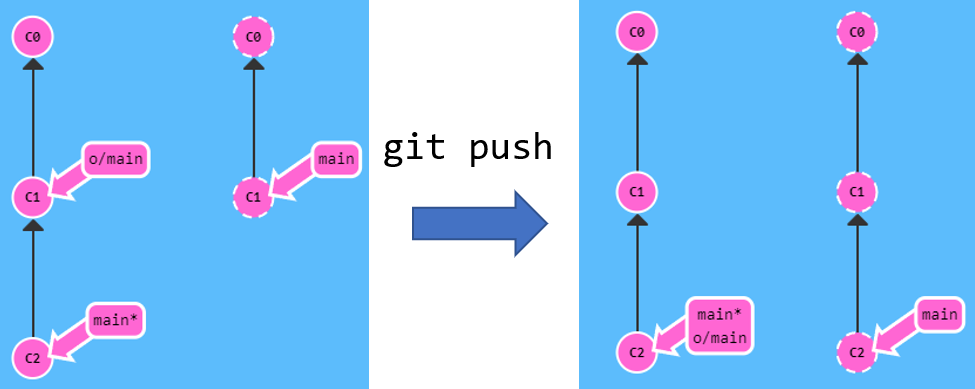


**Git Push**

**git push** is a way to upload the changes to a specified remote and incorporate the new commits.

In the example below, by doing git push:

* The remote received commit **C2** and the branch **main** was updated to point at **C2**
* The reflection of the remote **origin/main** was updated as well



**git push** only pushes the branch that is currently being targeted.

**Workflow example 1**

1. **Creating the working directory**

First thing to do is create a working directory (in this case in our Desktop), with the name “Story”.

**cd Desktop**

**mkdir Story**

**cd Story**

**touch chapter1.txt**

**touch chapter2.txt**

**touch chapter3.txt**

|  |  |
| --- | --- |
|  |  |

We can also add content to the files. To open the files use the start command:

|  |  |
| --- | --- |
|  |  |

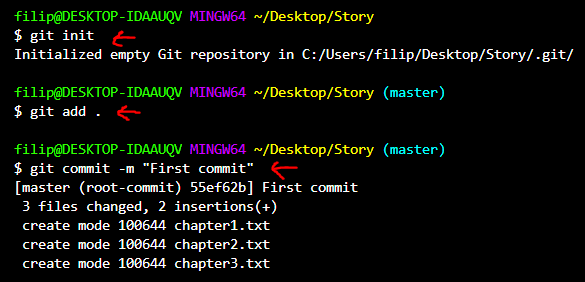
1. **Creating a local repository (.git)**

Now we need to create a local repository and add the files in the working directory to the repository. Also, we are going to do the first commit.

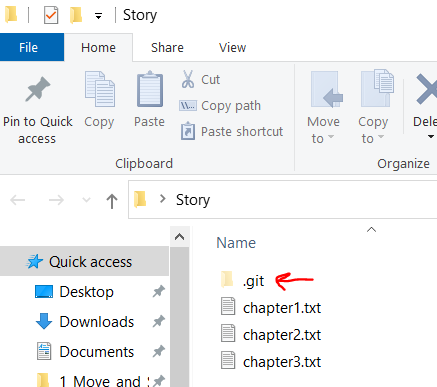
git init

git add .

git commit -m “First commit”



We now have a local repository called ***.git*** in our Working Directory which was originated on ***git init***.



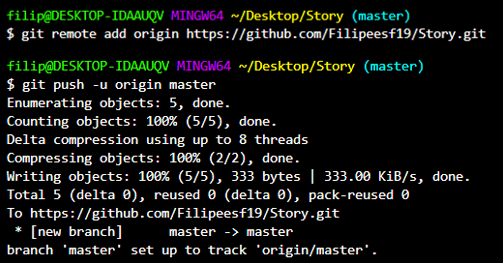
1. **Creating a Remote Repository and uploading files**

To create a Remote Repository you need to go to GitHub ([www.github.com](http://www.github.com)) and create a repository. You need an account to do this.

Then type the following code:

git remote add origin <https://github.com/Filipeesf19/Story.git>

git push -u origin master



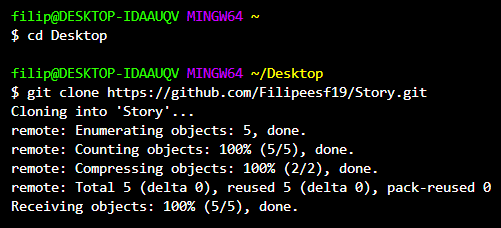
Now, if you visit your GitHub account, you can find your files there.



1. **Cloning the Remote Repository to your local machine**

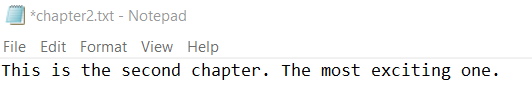
Now imagine you want to edit your project. You have to clone it, i.e. create a local copy of the project in your computer, just like downloading but with some differences. With cloning, you copy all the story as well, like commits. Remember to be at the Desktop directory so the story folder from your GitHub account is created there.

git clone <https://github.com/Filipeesf19/Story.git>



1. **Make changes to the file and create a second commit. Add it to the Remote Repository.**

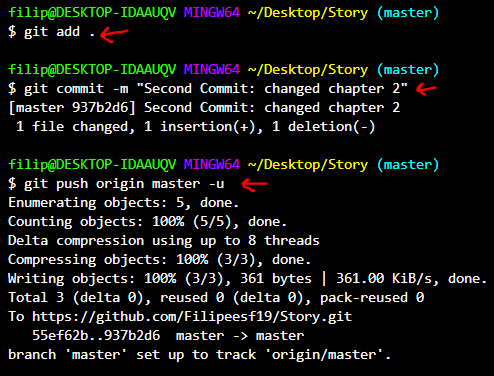
So we have made the changes in *chapter2.txt* as shown below.



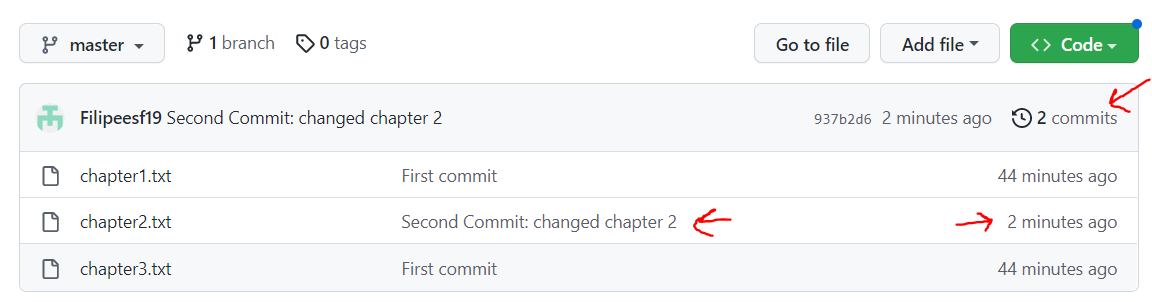
git add .

git commit -m “Second Commit: changed chapter 2”

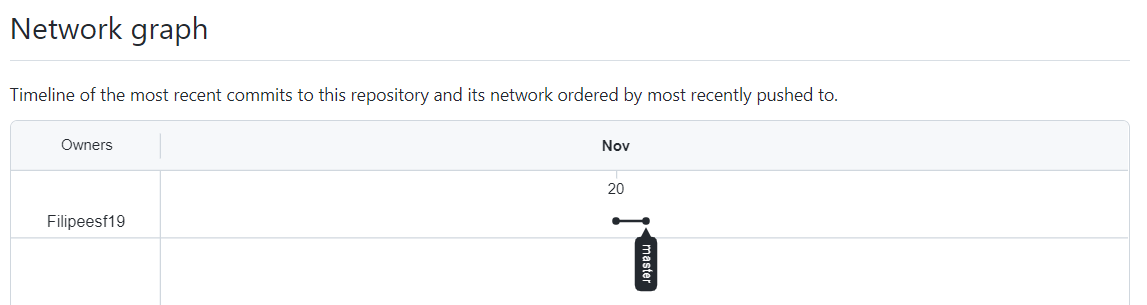
git push origin master -u

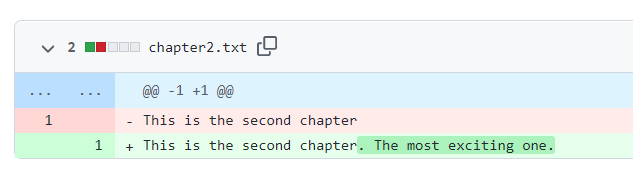


Now if you go back to your GitHub repository and refresh, you see chapter 2 has been updated and now shows 2 commits.



If you now go to *Insights 🡪 Network* you can see the story of commits in your project. You can also click on them and see the changes.



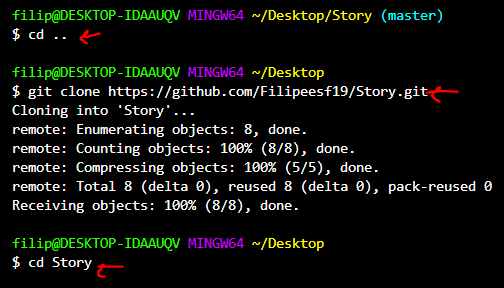


**Workflow example 2**

1. **Cloning the Remote Repository**

Bring the remote repository to your local environment so you can edit the project.

git clone <https://github.com/Filipeesf19/Story.git>

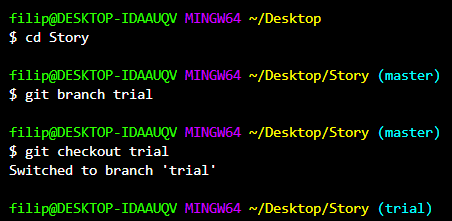


1. **Create a new branch called “trial” and add it to the local repository.**

In this step we not only create the branch “trial”, but we also do checkout to it, so we are now controlling it. Any changes we do to our files, are being done in the trial branch and won’t affect the master branch.

git branch trial

git checkout trial



1. **Change chapter 3 and create a chapter 4 in the “trial” branch**

Now let’s change the files in the “trial” branch and commit it.

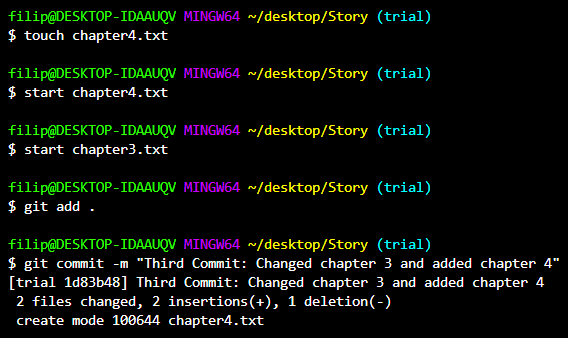
touch chapter4.txt

start chapter4.txt

start chapter3.txt

git add .

git commit -m “Third Commit: Changed chapter 3 and added chapter 4”



1. **Modify chapter 1 in the master branch**

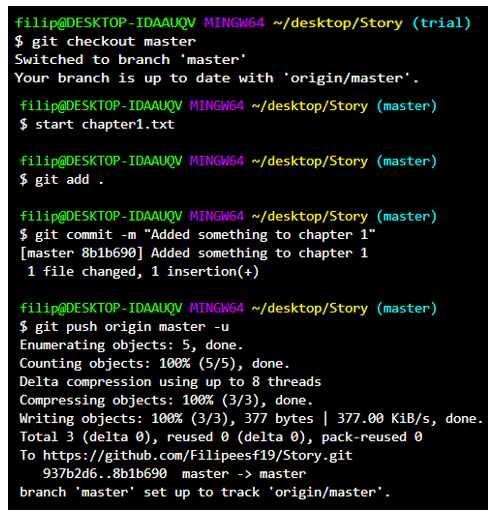
git checkout master

start chapter1.txt

git add .

git commit -m “Added something to chapter 1”

git push origin master -u

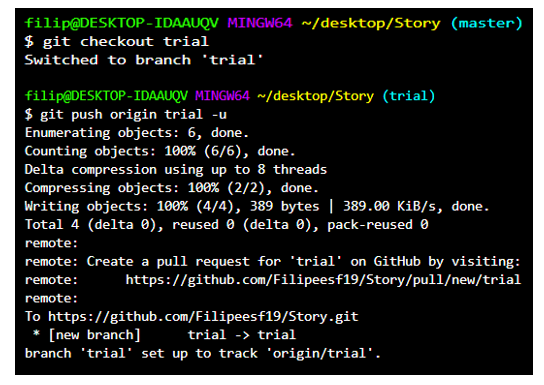


1. **Add the branch “trial” to the remote repository**

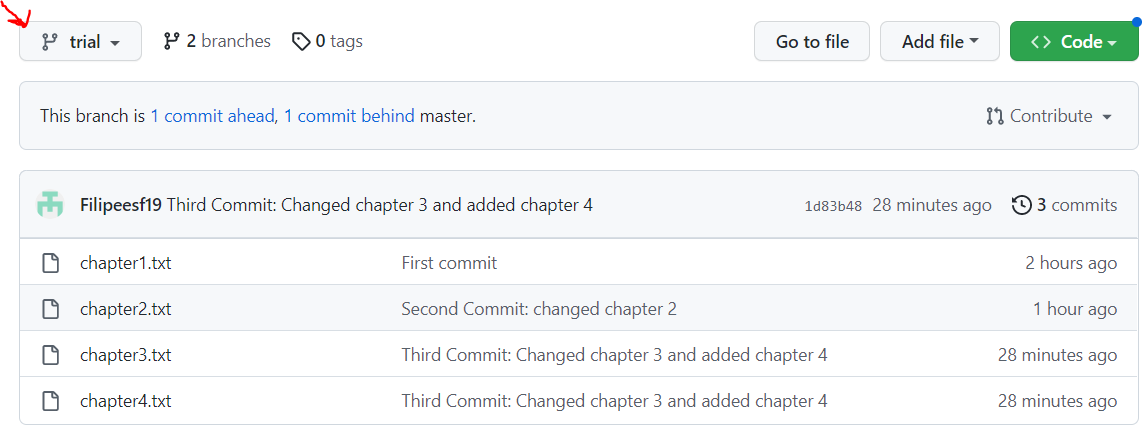
Let’s go back to the “trial” branch because we forgot to push it to the remote repository.

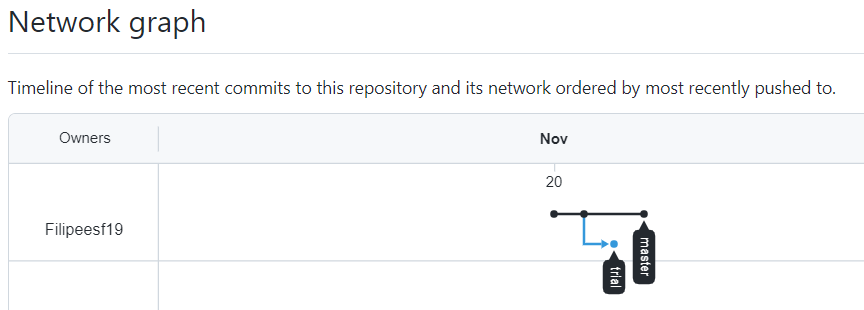
git checkout trial

git push origin trial -u

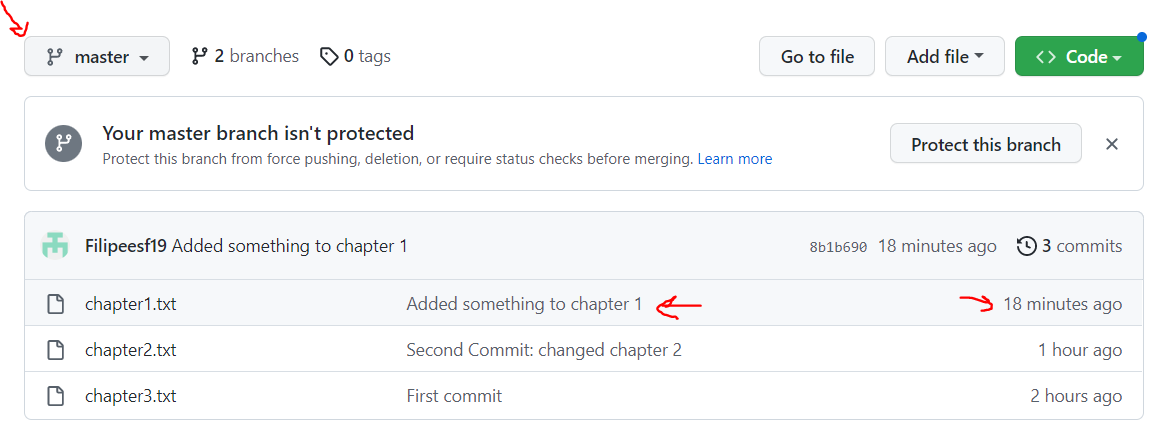


Now, if you go to the “trial” branch in your GitHub repository, you can see the new chapter and the modified chapter 2, but not the modified chapter 1.





But meanwhile in the “master” branch, you can see the modified version of chapter 1, but not the modified chapter 2 and added chapter 4, which was done on the “trial branch”.



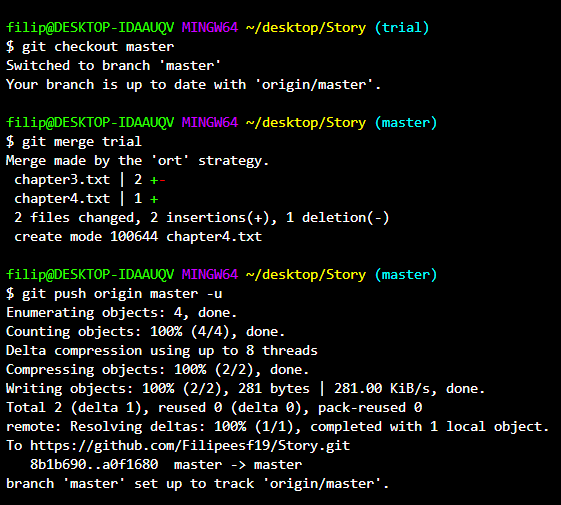
1. **Merging “trial” and “master”**

Now that we have checked that the modifications done in the “trial” branch were good, we will merge the “trial” into the “master” branch.

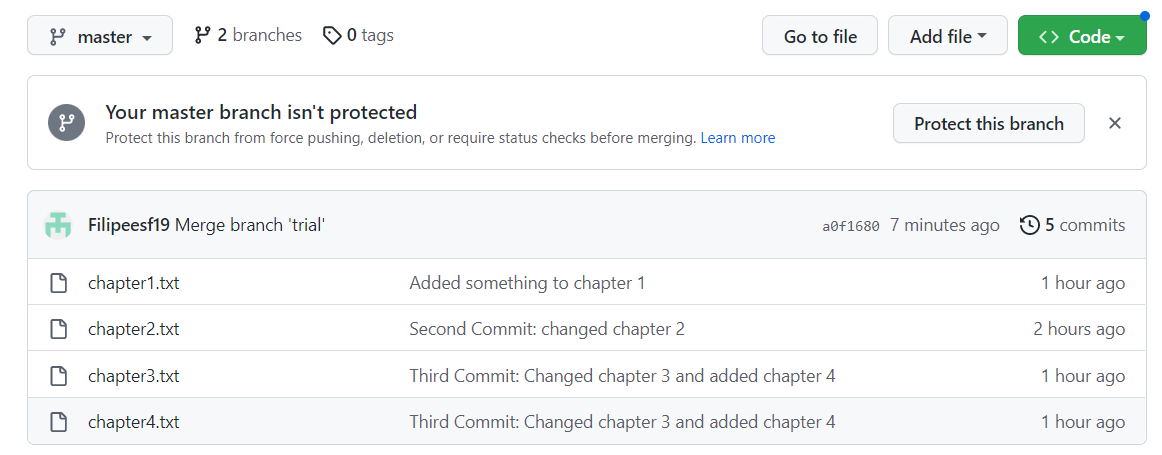
git checkout master

git merge trial

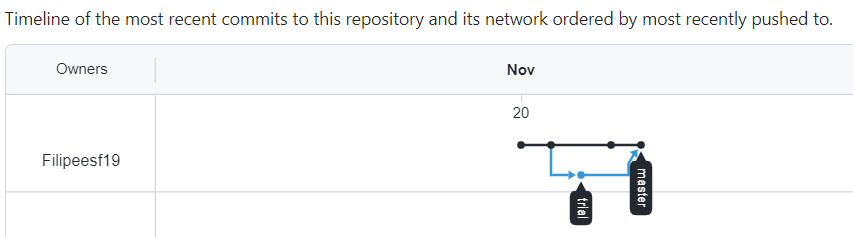
git push origin master -u



Notice now that, in your GitHub repository, in the “master” branch, you know have the changes done in both branches.



Also, the Network tree now includes the “trial” branch.

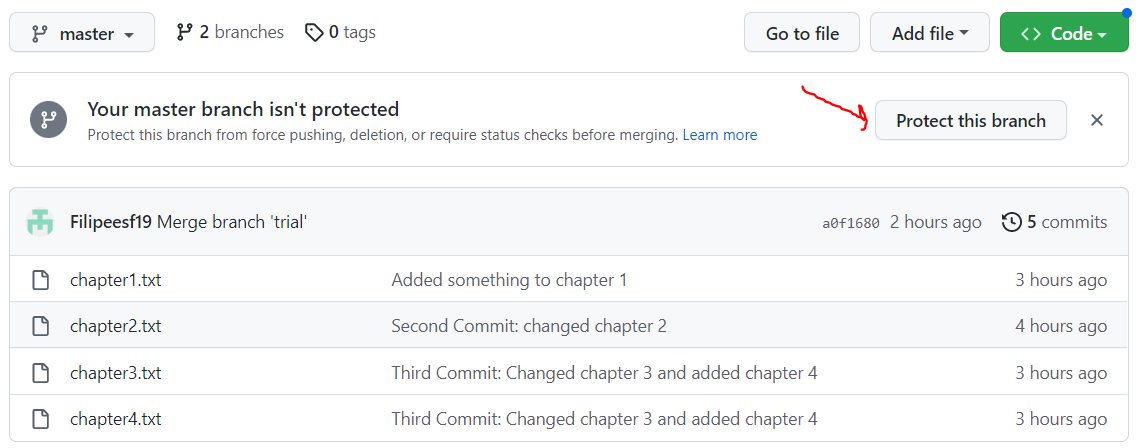


**Forking and Pull Requests**

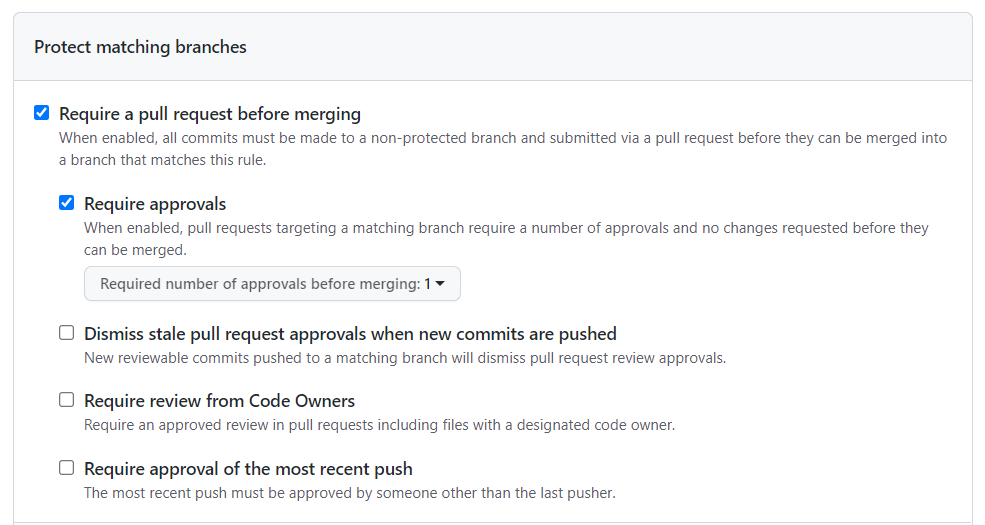
A fork in Git is simply a **copy** of an **existing** **repository** in which the **new** **owner** **disconnects** **the** **codebase** **from** **previous** **committers**.

Most commonly, **forks** are used to either **propose changes to someone else's project** to which you do **not have write access**, or to **use someone else's project as a starting point for your own idea**. You can fork a repository to create a copy of the repository and make changes **without affecting the upstream repository**.

You can make your project available to everyone, or maybe just available to some people (for example if you work in a team and all in the same project), or maybe not available to anyone. The point is, if you don’t have write access, you have to send a **pull** **request**.



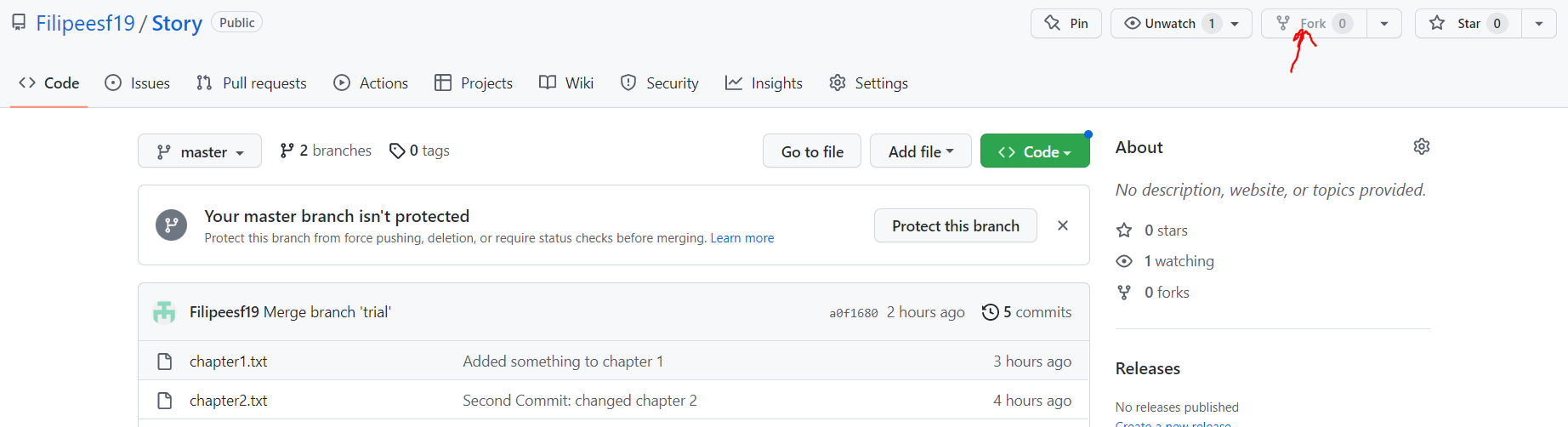
There are various levels of protection you can set to a branch, as seen in the image below.



So, let’s say I see someone’s code on GitHub that I want to improve:

1. I **fork** it to my **remote** **repository**.
2. I **clone** it to my **local** **environment**.
3. Implement the **changes.**
4. **Push** it to my **remote** **repository**.
5. Send a **pull** **request** (the two repositories will be merged).

To fork a project, click in the “Fork” button in the right-hand corner of the repository page.



Once the changes are done, you can go to the repository page and click “New Pull Request”.

The project owner can now go to the “Pull requests” tab and approve it.

**Moving around the commit tree**

it's important to understand different ways to move through the commit tree that represents your project.

**HEAD** is the symbolic name for the **currently checked out commit** - it's essentially what commit you're working on top of.

HEAD **points to the most recent commit which is reflected in the working tree by default**. Most git commands which make changes to the working tree will start by changing HEAD.

**Detaching HEAD**

**Detaching** **HEAD** just means **attaching it to a commit** instead of a branch.

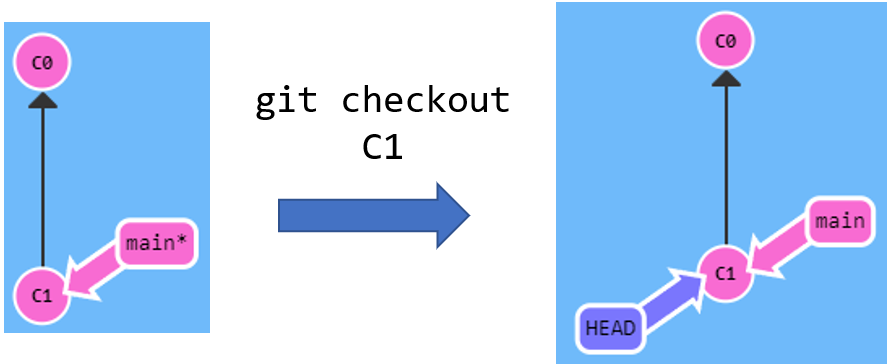
This is what it looks like beforehand:

**HEAD -> main -> C1**

And then it’s:

**HEAD -> C1**

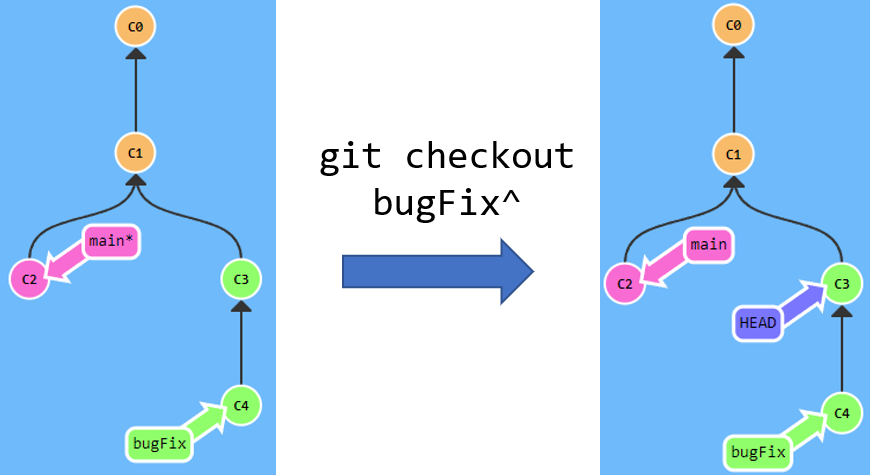
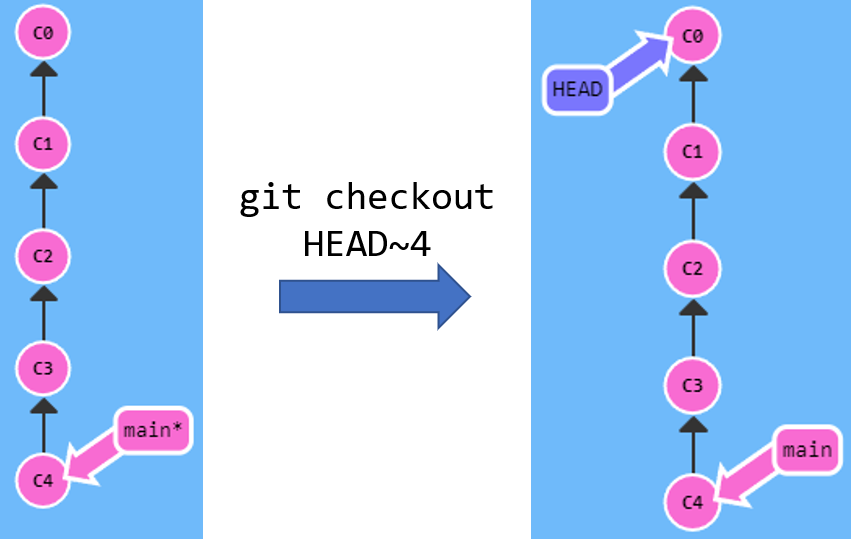
**main -> C1**



**Relative Refs**

With **relative refs**, you can start somewhere memorable (like the branch **bugFix** or **HEAD**) and work from there. They allow to:

* Move upwards one commit at a time with **^**
* Move upwards a number of times with **~<num>**

**Absolute Refs**

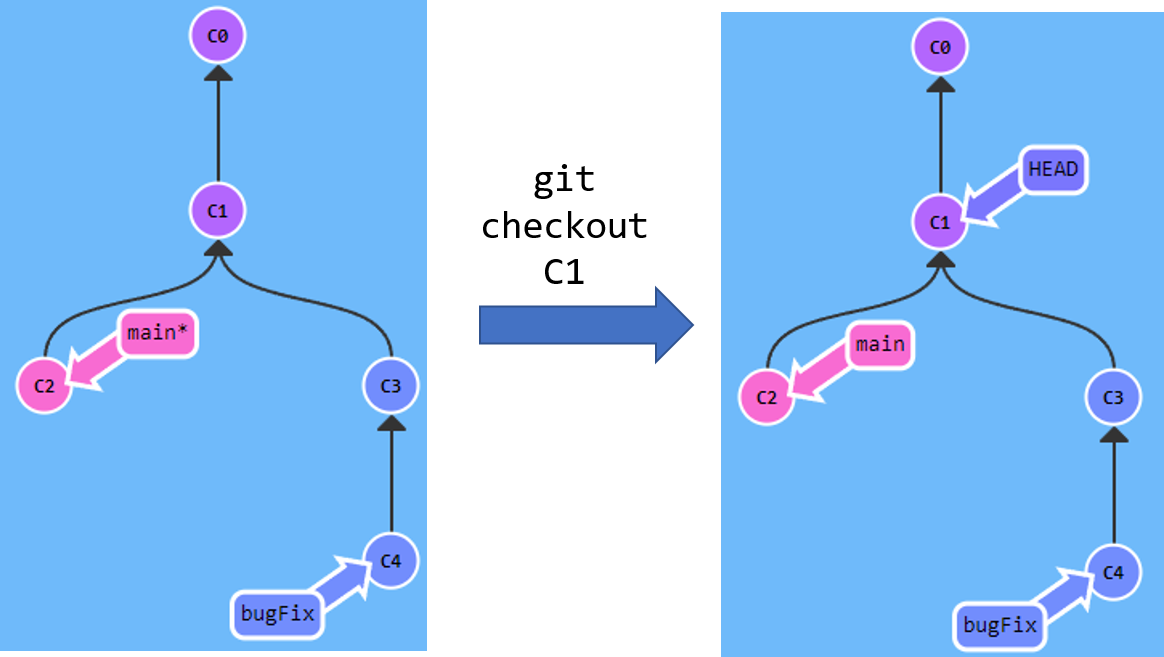
You can also use absolute refs so your HEAD goes to a specified commit.

In this example, and to simplify, we simply refer to the First commitement as C1. In reality, you have to refer by the hash. The hash number can be found using git log.

The hashes are nasty numbers like **fed2da64c0efc5293610bdd892f82a58e8cbc5d8**.

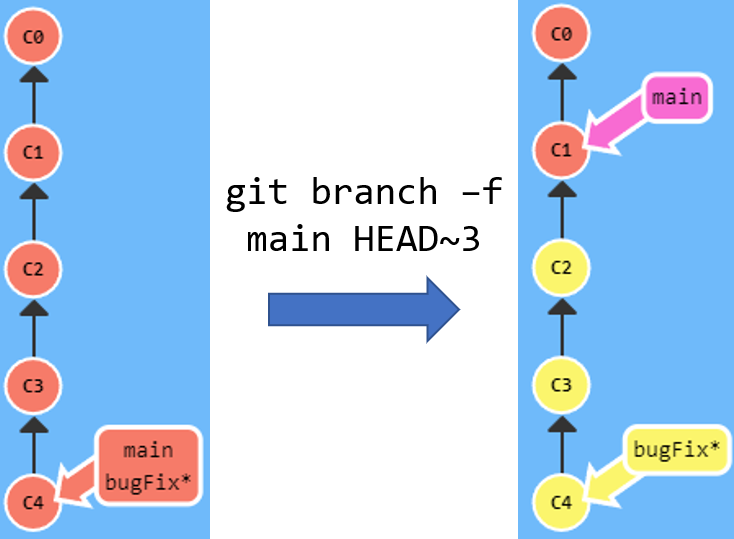
The upside is that Git is smart about hashes. It only requires you to specify enough characters of the hash until it uniquely identifies the commit. So I can type **fed2** instead of the long string above.

To see the the hashes you have use the **git log** command.

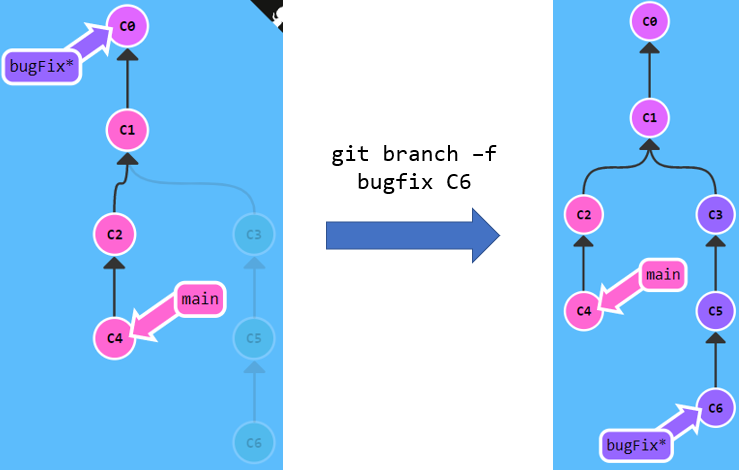


**Moving branches to other commits**

One of the most common ways I use relative refs is to move branches around. You can directly reassign a branch to a commit with the **-f** option.



You can also use absolute refs as well (the hash ref of a certain commit) to move the branch to that commit. Again, in the example below, we are using C6 instead for ref. instead of the actual hash for simplification.



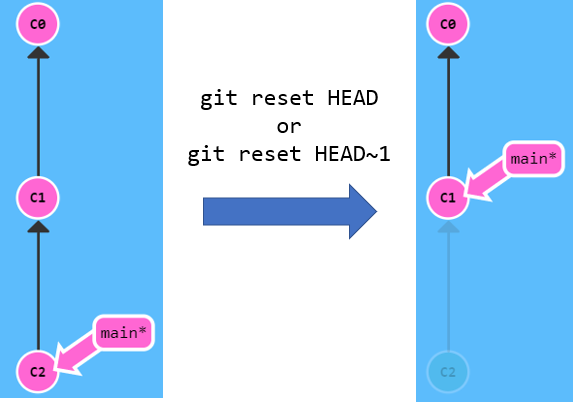
**Reversing Changes**

There are two primary ways to undo changes in Git -- one is using **git reset** and the other is using **git revert**.

**Git Reset**

**git reset** is a powerful command that is used to **undo local changes**. It is a simple way to undo changes that **haven’t been shared with anyone else**.

**git reset** will move a branch backwards as if the commit had never been made in the first place. In the example below, Git moved the main branch reference back to **C1**, so now the local repository is in a state as if **C2** had never happened.



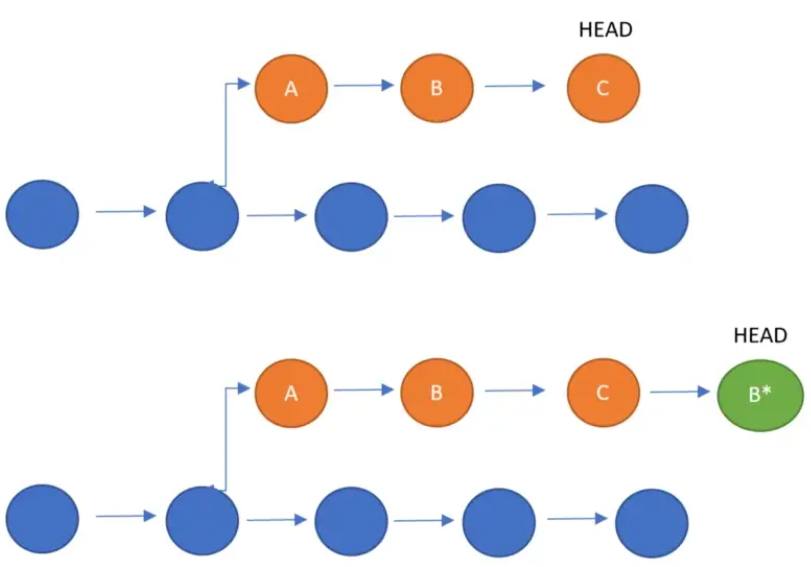
**Git Revert**

While resetting works great for local branches on your own machine, its method of "rewriting history" doesn't work for remote branches that others are using.

In order to **reverse changes and share those reversed changes with others**, we need to use **git revert**.

**git revert** actually creates a new commit. This commit will change the state as it was in the previous commit. For instance, in the example below, we were in commit **C**, and after reverting, it created a new commit **B\***, which is essentially commit **B**.

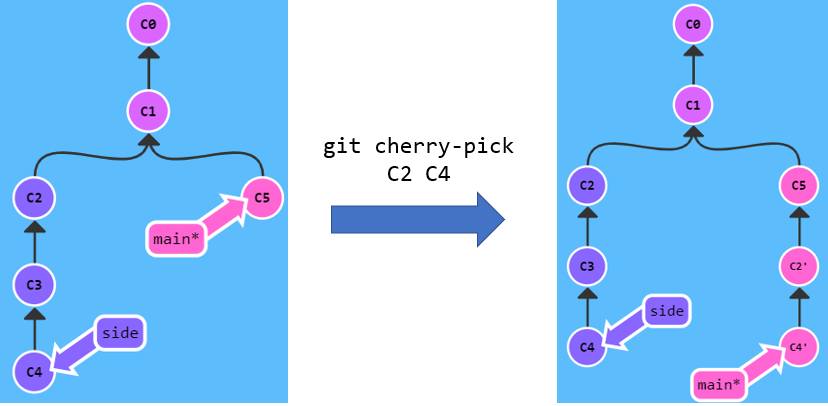
This **preserves the commit history** and potentially reduces the amount of merge conflicts that your team may have to deal with.



**Cherry Picking**

**git cherry-pick** is a very straightforward way of saying that you would like to **copy a series of commits below your current location** (**HEAD**).

In the example below, we have used C2 and C4 for simplification, but you would have to use the commit hashes.



**Diverged Work**

So far we've seen how to **pull** down commits from others and how to **push** up our own changes. It seems pretty simple, so how can people get so confused?

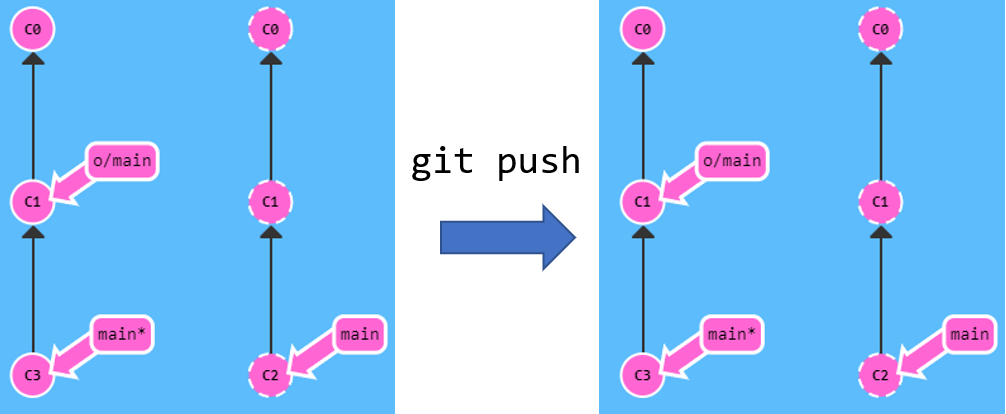
The difficulty comes in when the history of the repository diverges.

Imagine the following situation:

* On Monday, you clone a repository.
* By Friday you are ready to publish (push) your work back into the repository, but your coworkers already written a bunch of code during that week that has made your changes our of date.

Git won’t allow you to run **git push** in these case where history has diverged. **It forces you to incorporate the latest state of the remote before being able to share your work**.

In the example, nothing happened because the command fails. **git push** fails because your most recent commit **C3** is based off of the remote at **C1**. The remote has since been updated to **C2** though, so git rejects your **push**.

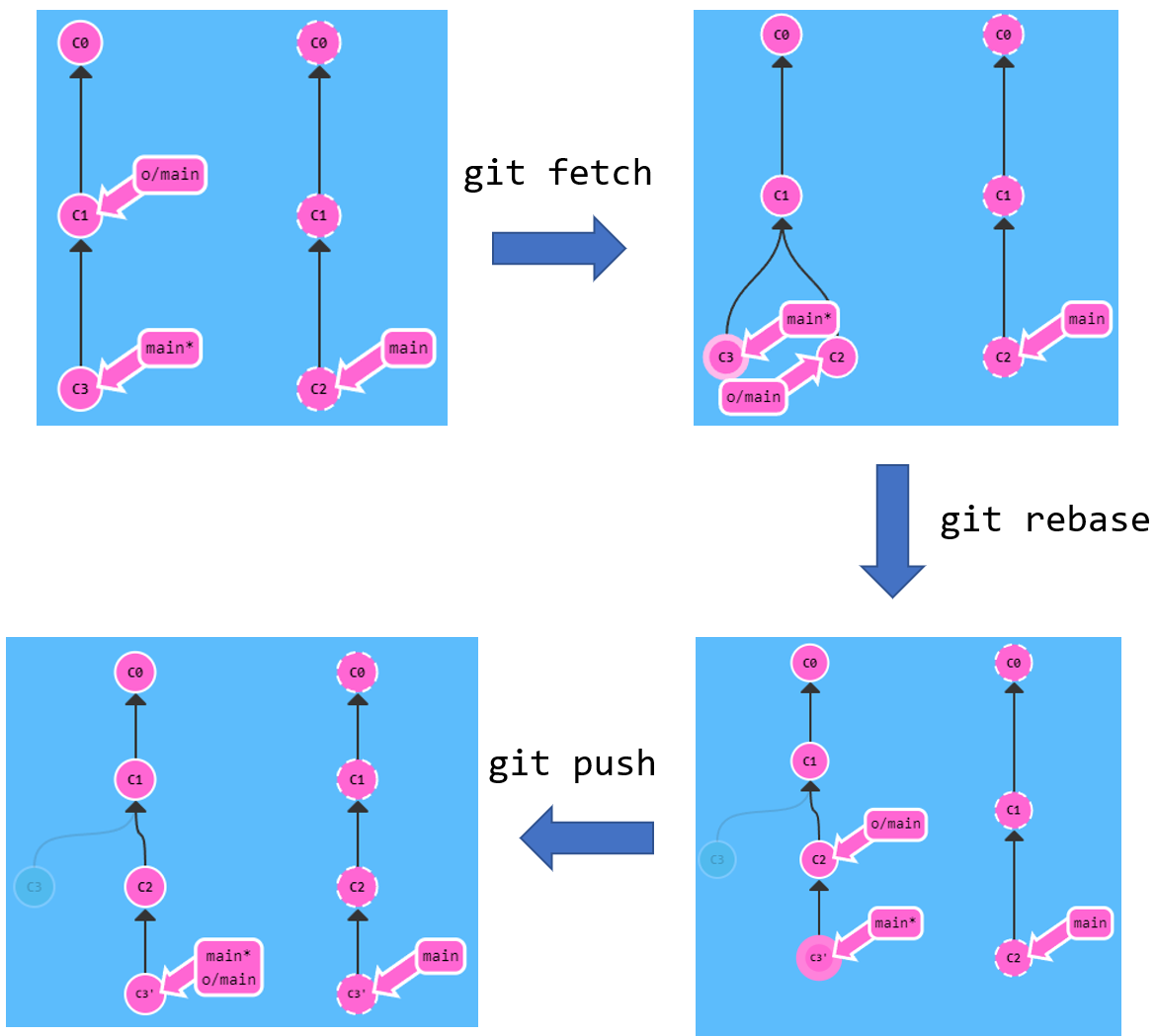


All you need to sort this problem is **base your work off of the most recent version of the remote branch**.

**Rebasing**

One way to sort this problem is by rebasing. In the example below, we:

1. updated our local representation of the remote with **git fetch**
2. rebased our work to reflect the new changes in the remote with **git rebase**
3. pushed them with **git push**.

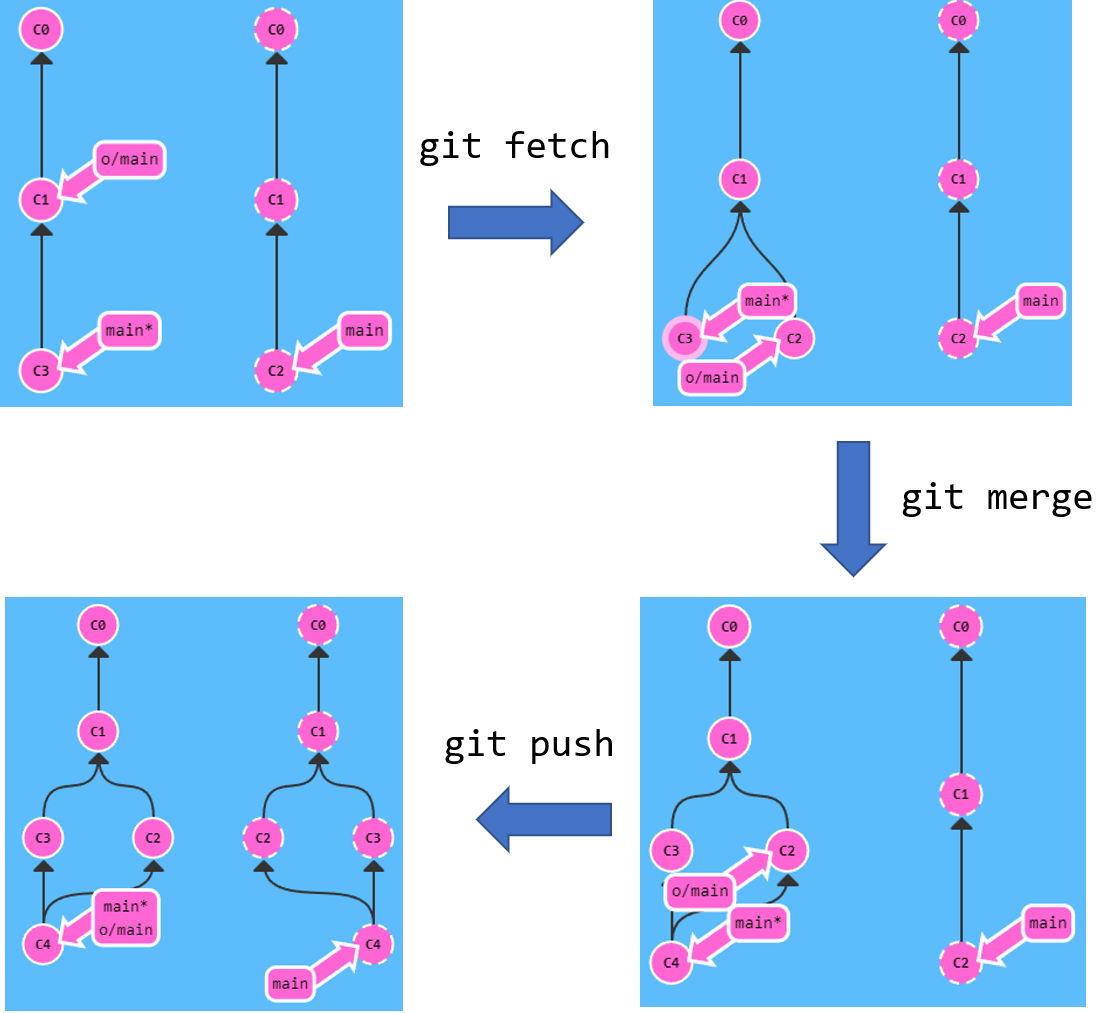


**Merging**

Another way to sort this problem is by rebasing. In the example below, we:

1. updated our local representation of the remote with **git fetch**
2. merged the new work into our work (to reflect the new changes in the remote) with **git merge**
3. pushed them with **git push**.

Although **git merge** doesn't move your work like **git rebase** - and instead just **creates a merge commit** - it's a way to tell git that you have incorporated all the changes from the remote. This is because the **remote branch is now an ancestor of your own branch**, meaning your **commit reflects all commits in the remote branch**.

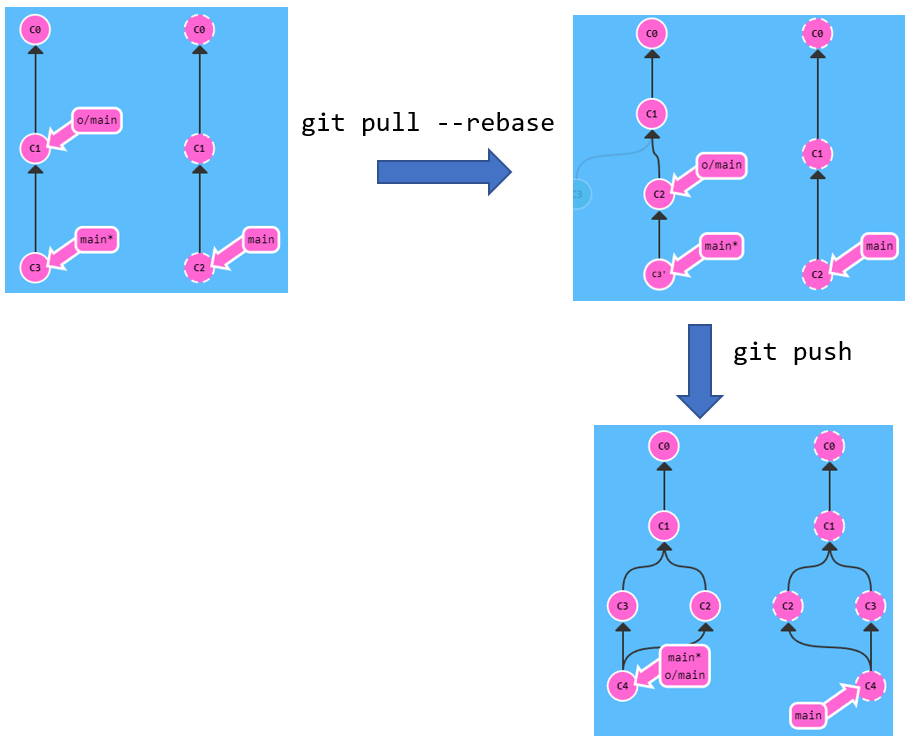


**Pull and rebasing**

The third way to sort this is by doing **git pull --rebase**, which is shorthand for a **git fetch** and a **git rebase**!

In the example below, we:

1. updated our local representation of the remote and rebased our work to reflect the new changes in the remote with **pull --rebase**.
2. pushed them with **git push**.



**Pull and Push**

Another way to do it is by doing **pull** and **push**. Which is pretty much like **fetching** and **merging**, but all the one command.

