Git Commands

Git is a version control system that allows developers to collaborate on a codebase and track changes to files over time. Here are some of the most commonly used Git command lines with their explanations:

`git init`: This command initializes an empty Git repository in the current directory.

`git clone <repository-url>`: This command creates a copy of an existing Git repository in a new directory.

`git add <file>`: This command adds a file to the staging area, which means it's ready to be committed.

`git commit -m "Commit message"`: This command saves the changes in the staging area to the Git repository with a descriptive commit message.

`git push`: This command sends committed changes from the local repository to a remote repository.

`git pull`: This command retrieves the latest changes from a remote repository and merges them into the local repository.

`git status`: This command shows the current status of the repository, including which files are staged for commit and which have changes that are not yet staged.

`git log`: This command shows a history of all the commits made to the repository, including the commit message, author, and timestamp.

`git branch`: This command lists all the branches in the repository and highlights the currently active branch.

`git checkout <branch-name>`: This command switches to a different branch in the repository.

`git merge <branch-name>`: This command merges the changes made in a specified branch into the current branch.

`git stash`: This command temporarily saves changes that are not yet ready to be committed, so that you can switch to a different branch or work on a different task without losing your changes.

`git reset <file>`: This command removes a file from the staging area, but keeps the changes in the working directory.

`git revert <commit>`: This command creates a new commit that undoes the changes made in a specified commit.

`git remote add <name> <url>`: This command adds a remote repository with the specified name and URL to the local repository.

These are just a few of the most commonly used Git command lines. Git has many other commands and options that allow for fine-grained control over the version control process.

How to create a new branch

Ok! You are all ready to get branching. Once this window closes, make a new branch named `bugFix` and switch to that branch.

By the way, here's a shortcut: if you want to create a new branch AND check it out at the same time, you can simply type `git checkout -b [yourbranchname]`.

`git branch < New branch name >`

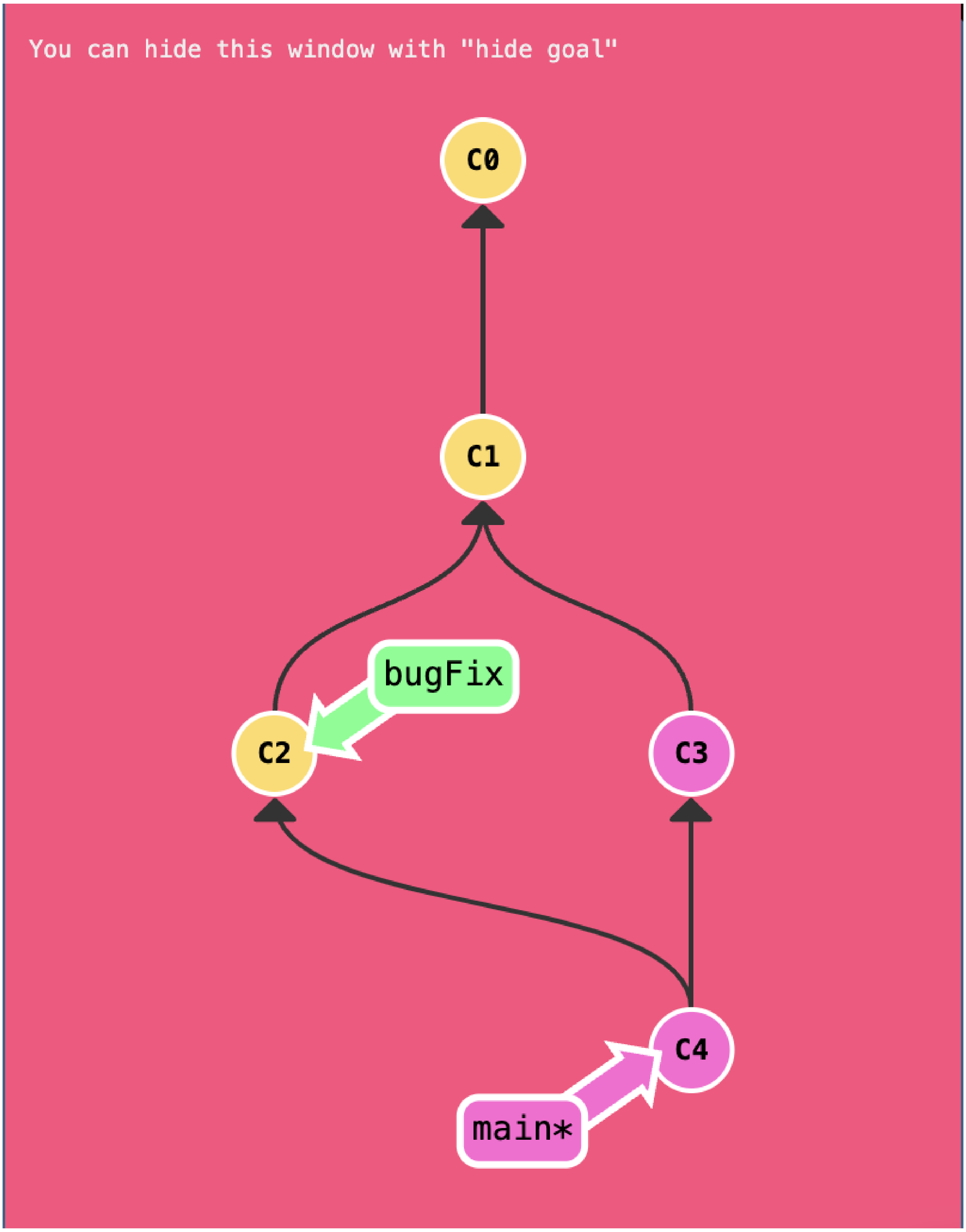
How to switch branch

`git checkout < Branch name >`

How to merge two branches

`git merge < Branch name >` It will add the named branch to the currently postitioned branch.

```
git branch bugFix
git checkout bugFix
git commit
git checkout main
git commit
git merge bugFix
```

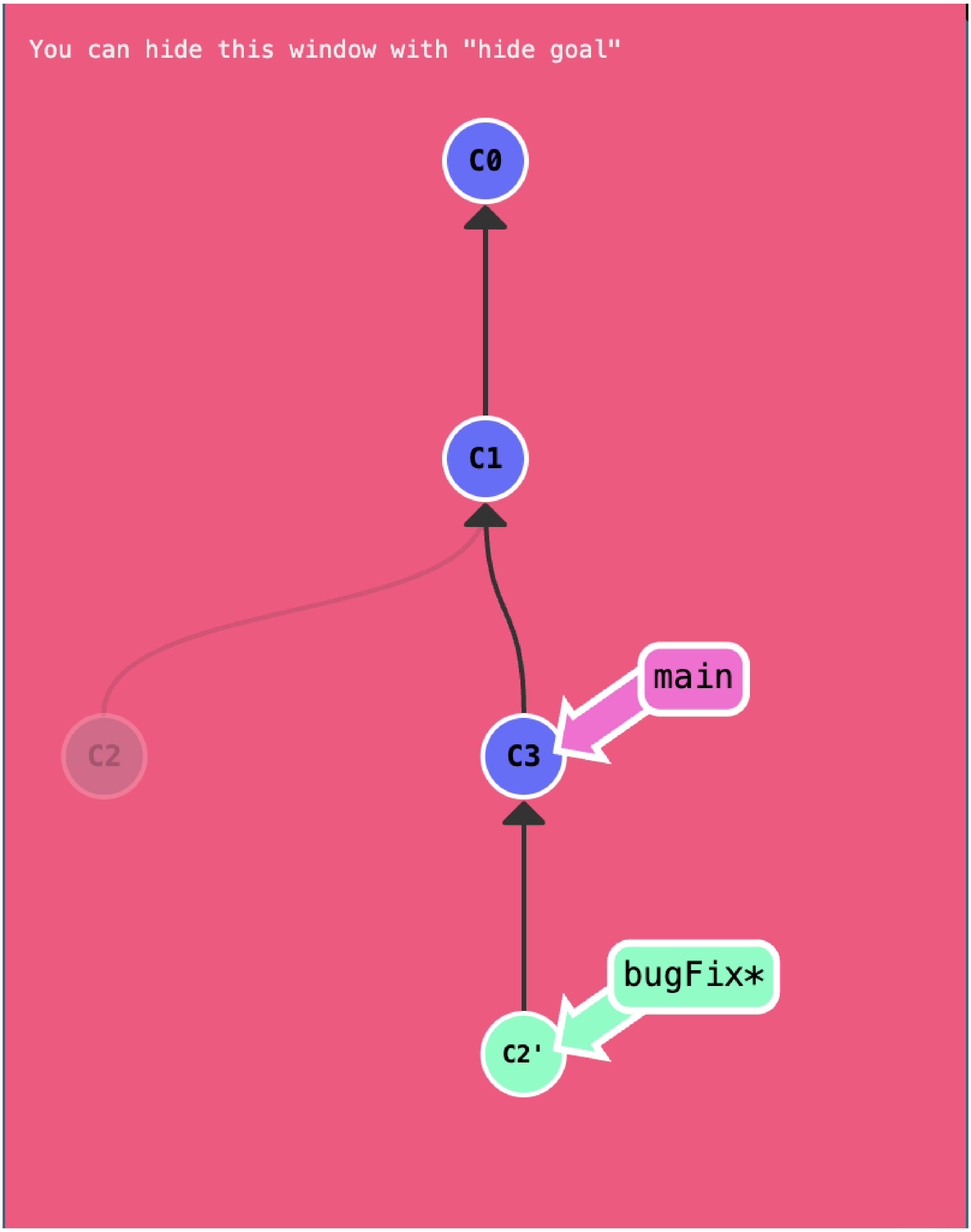


Hwo to rebase two branch

`git rebase < Branch name >` , It will rebase to the named brach from the currently postioned branch

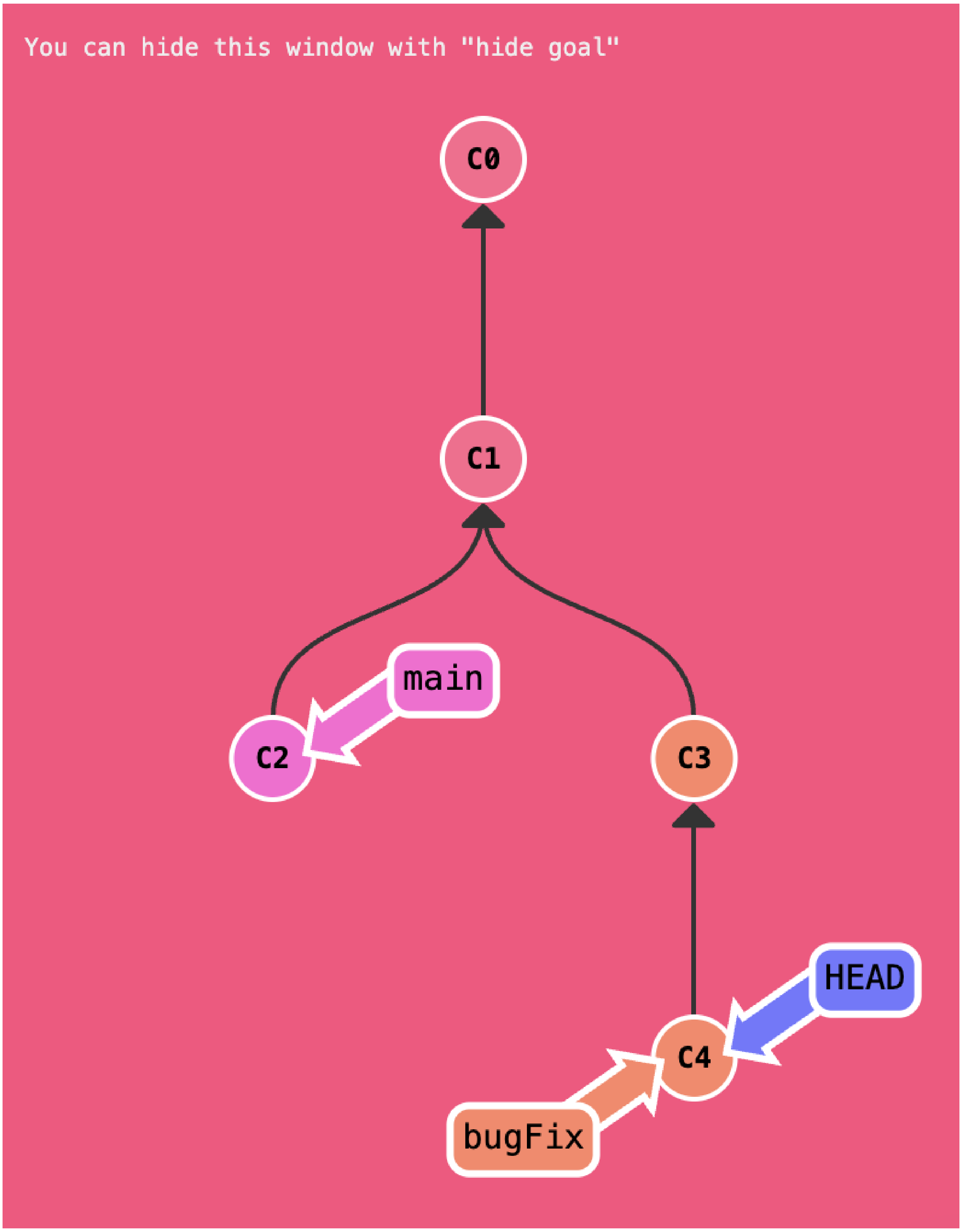
Example

```
git branch bugFix
git checkout bugFix
git commit
git checkout main
git commit
git checkout bugFix
git rebase main
```


You can hide this window with "hide goal"

Detach yo Head

```
git checkout C4
```



Relative Refs

Moving around in Git by specifying commit hashes can get a bit tedious. In the real world you won't have a nice commit tree visualization next to your terminal, so you'll have to use `git log` to see hashes.

Furthermore, hashes are usually a lot longer in the real Git world as well. For instance, the hash of the commit that introduced the previous level is `fed2da64c0efc5293610bdd892f82a58e8cbc5d8`. Doesn't exactly roll off the tongue...

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.

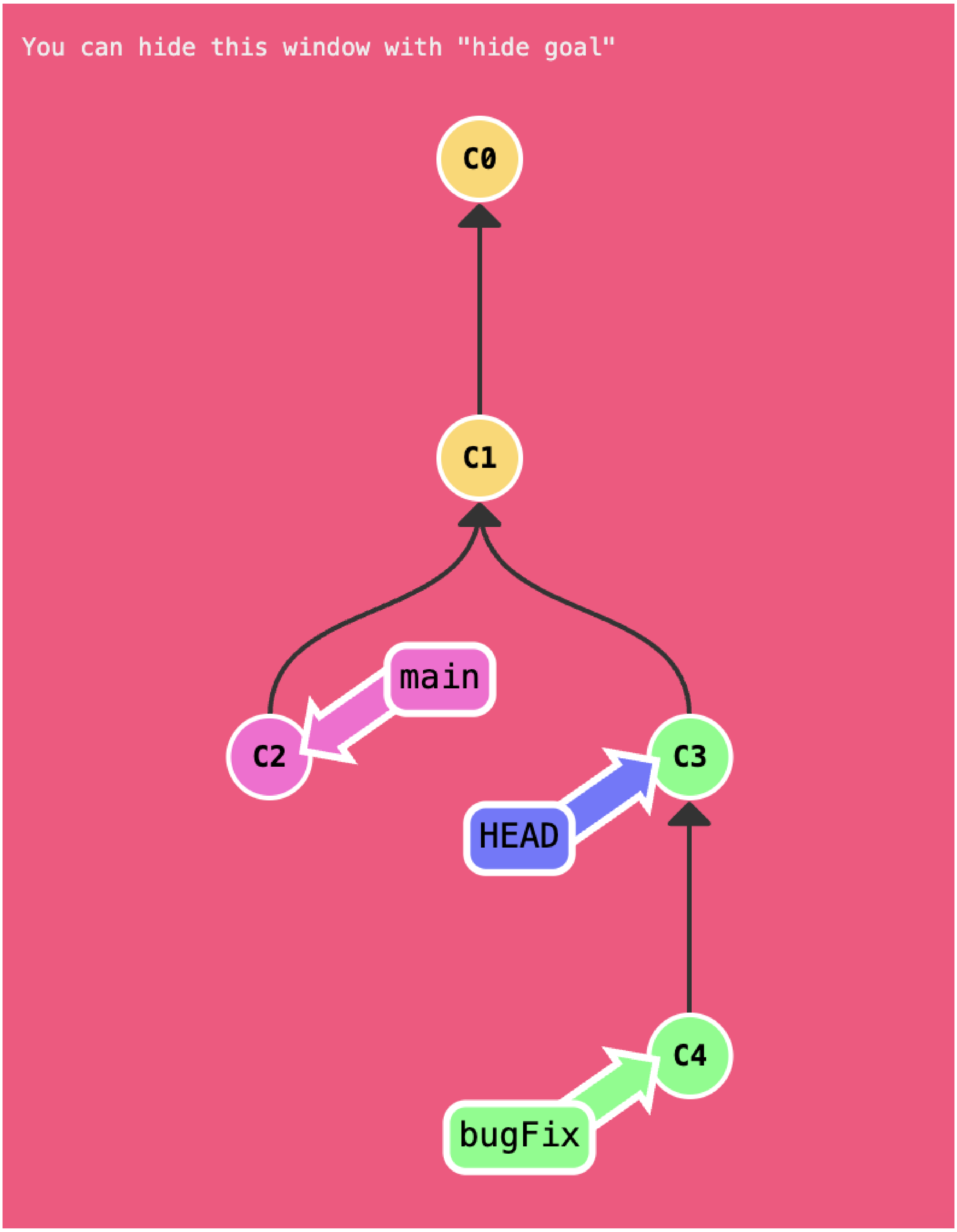
Like I said, specifying commits by their hash isn't the most convenient thing ever, which is why Git has relative refs. They are awesome!

With relative refs, you can start somewhere memorable (like the branch `bugFix` or `HEAD`) and work from there.

Relative commits are powerful, but we will introduce two simple ones here:

- Moving upwards one commit at a time with `^`
- Moving upwards a number of times with `~<num>`

```
git checkout c3
```



The "~" Operator

Say you want to move a lot of levels up in the commit tree. It might be tedious to type `^` several times, so Git also has the tilde (~) operator.

The tilde operator (optionally) takes in a trailing number that specifies the number of parents you would like to ascend. Let's see it in action.

Branch Forcing

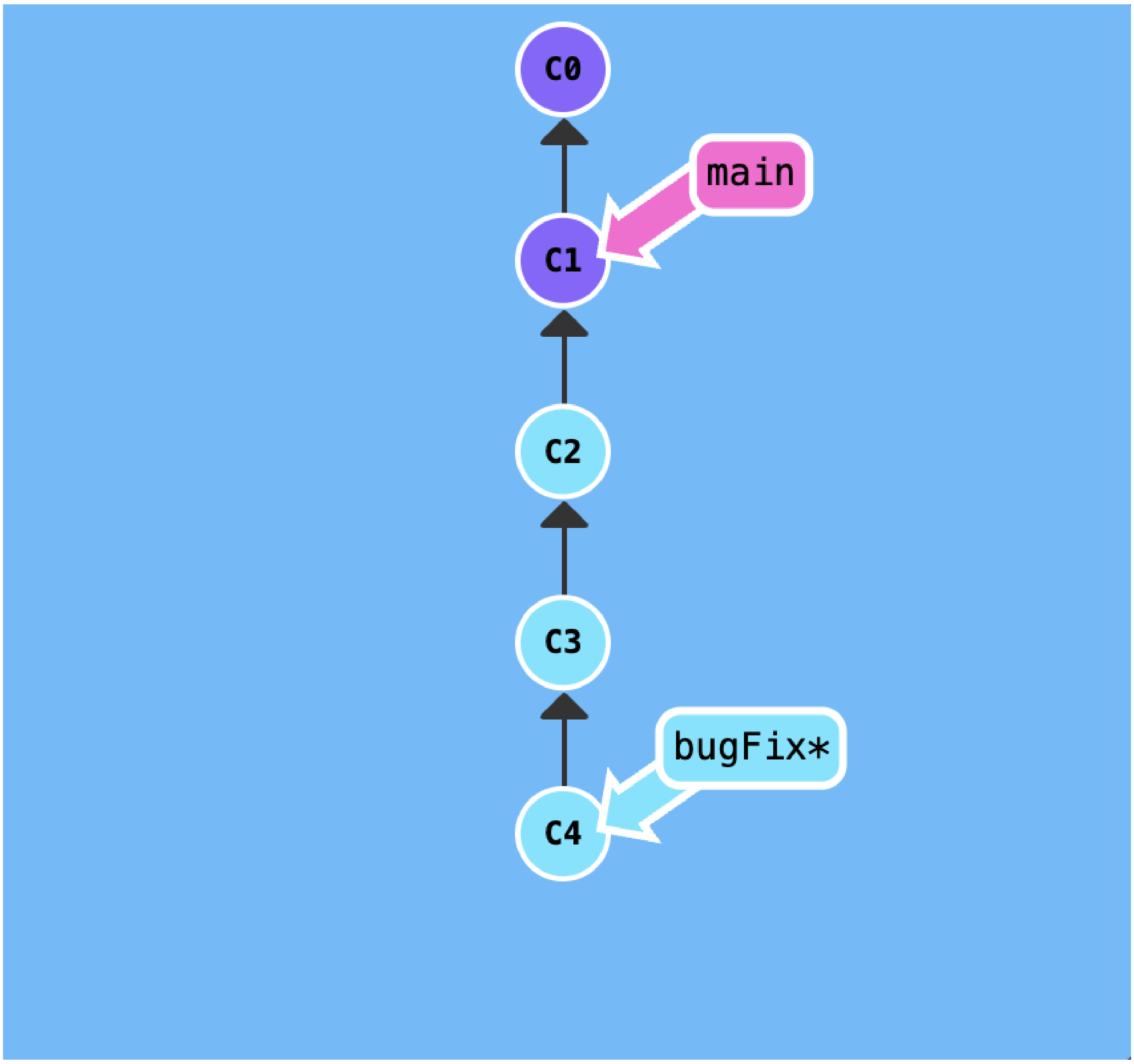
You're an expert on relative refs now, so let's actually *use* them for something.

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. So something like:

`git branch -f main HEAD~3`

moves (by force) the main branch to three parents behind HEAD.

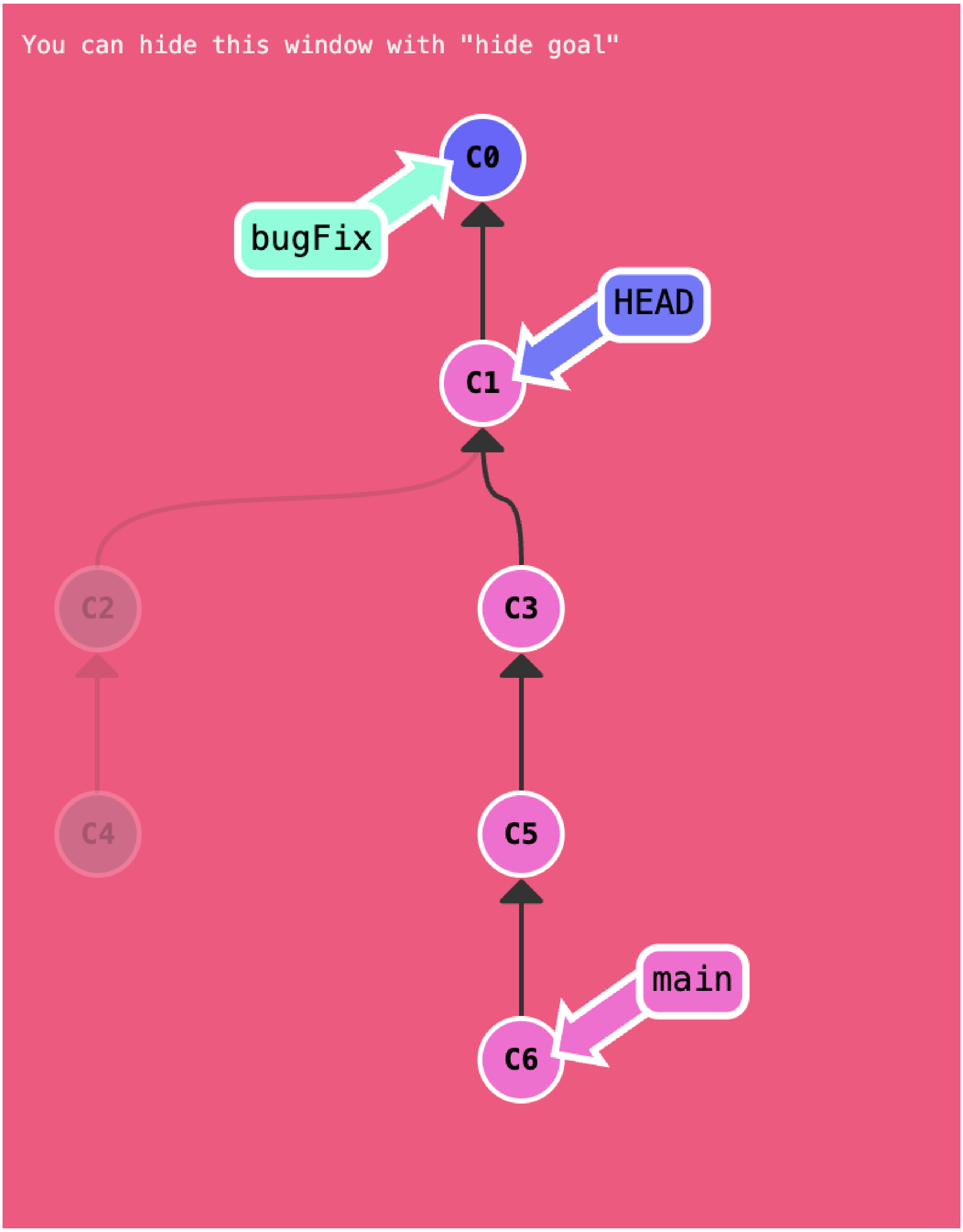
`git branch -f main HEAD~3`



challenge

Solution

```
git branch -f main C6
git branch -f bugFix C0
git checkout C1
```



You can hide this window with "hide goal"

Reversing Changes in Git

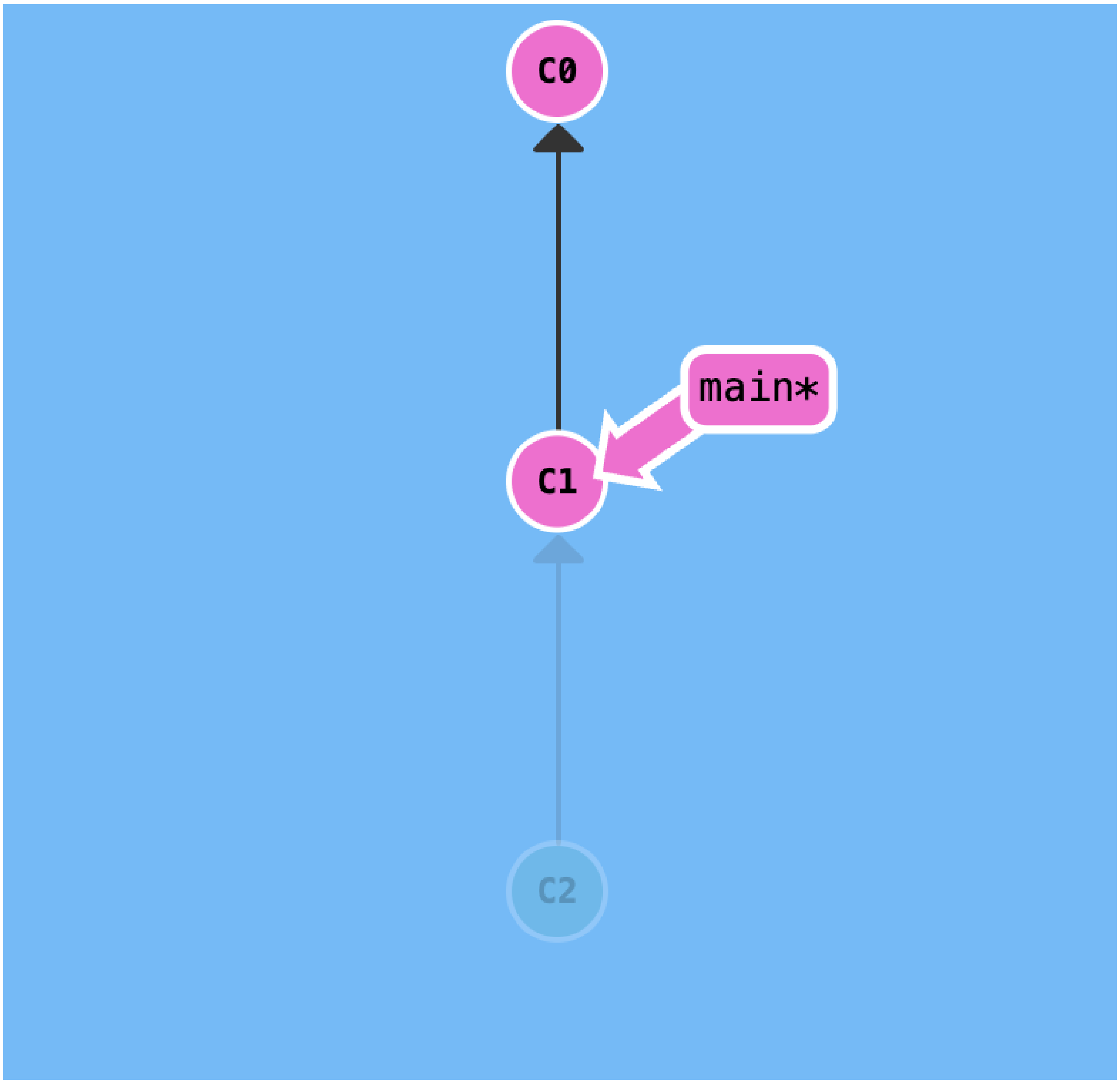
There are many ways to reverse changes in Git. And just like committing, reversing changes in Git has both a low-level component (staging individual files or chunks) and a high-level component (how the changes are actually reversed). Our application will focus on the latter.

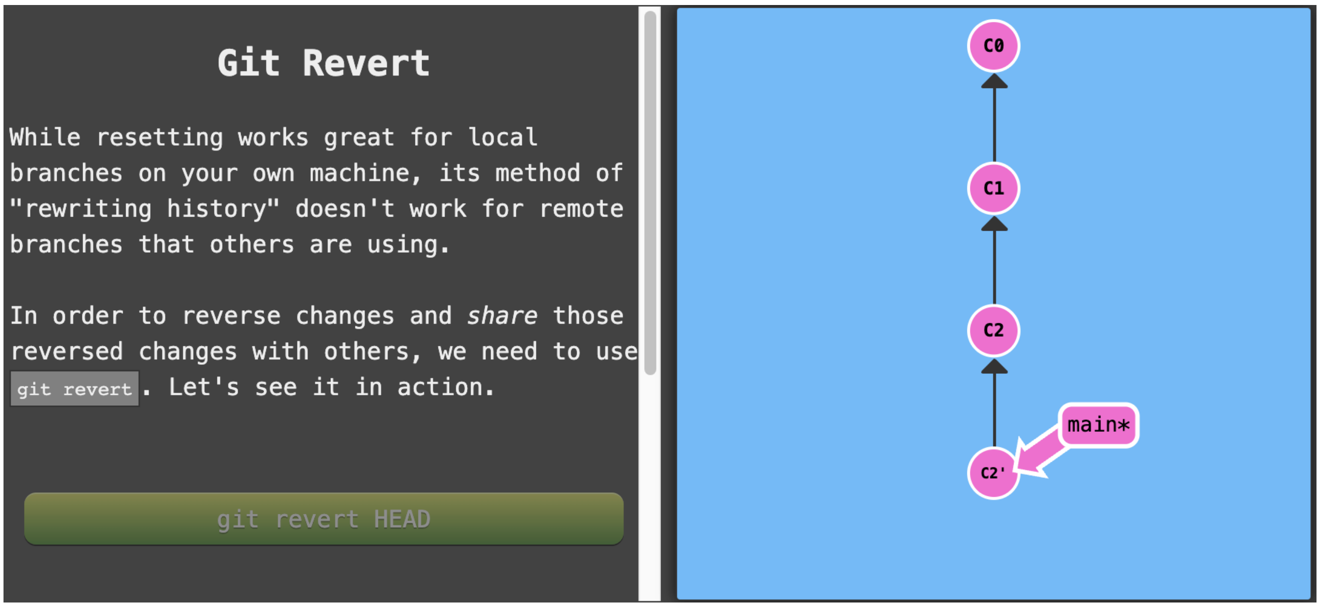
There are two primary ways to undo changes in Git -- one is using `git reset` and the other is using `git revert`. We will look at each of these in the next dialog

---

`git reset` reverses changes by moving a branch reference backwards in time to an older commit. In this sense you can think of it as "rewriting history;" `git reset` will move a branch backwards as if the commit had never been made in the first place.

`git reset HEAD~1`





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.

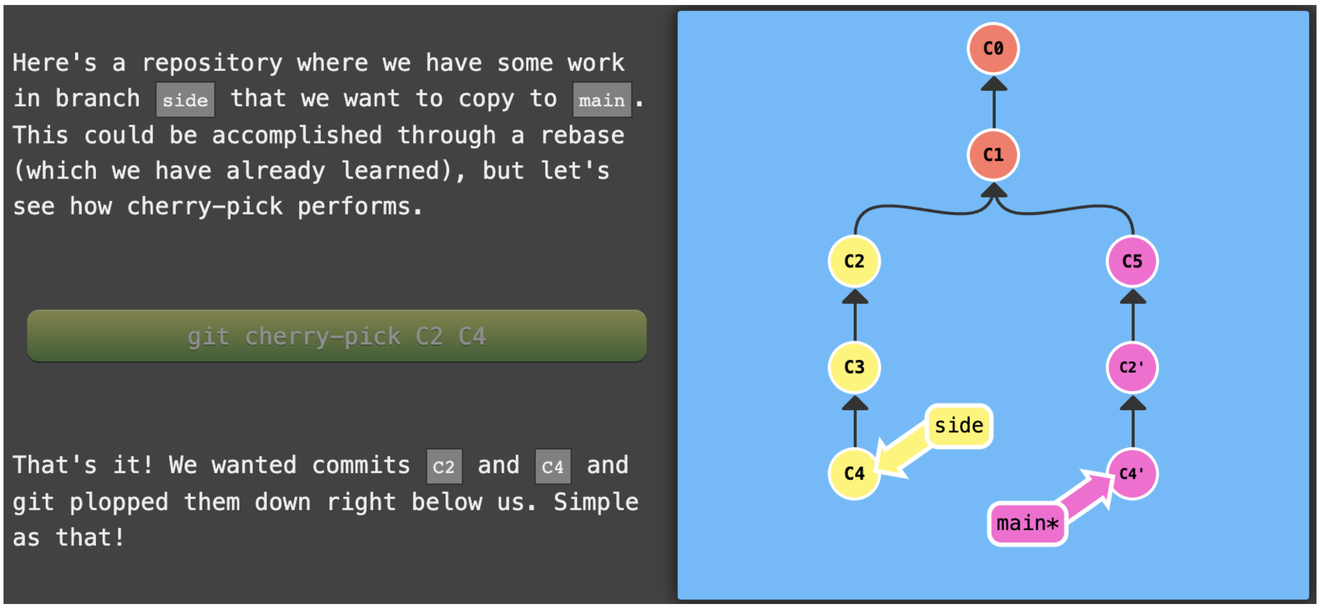
In order to reverse changes and *share* those reversed changes with others, we need to use `git revert`. Let's see it in action.

Git Cherry-pick

The first command in this series is called `git cherry-pick`. It takes on the following form:

- `git cherry-pick <Commit1> <Commit2> <...>`

It's a very straightforward way of saying that you would like to copy a series of commits below your current location (`HEAD`). I personally love `cherry-pick` because there is very little magic involved and it's easy to understand.



Here's a repository where we have some work in branch side that we want to copy to main. This could be accomplished through a rebase (which we have already learned), but let's see how cherry-pick performs.

```
git cherry-pick C2 C4
```

That's it! We wanted commits C2 and C4 and git plopped them down right below us. Simple as that!

Solution

`git cherry-pick c3 c4 c7`

Learn Git Branching Challenge Solutions

1.1 Introduction to Git Commits

git commit
git commit

1.2 Branching in Git

git branch bugFix
git checkout bugFix

1.3 Merging in Git

git checkout -b bugFix
git commit
git checkout master

```
git commit
git merge bugFix
```

1.4 Rebase Introduction

```
git checkout -b bugFix
git commit
git checkout master
git commit
git checkout bugFix
git rebase master
```

2.1 Detach yo' HEAD

```
git checkout C4
```

2.2 Relative Refs (^)

```
git checkout C4^
```

2.3 Relative Refs #2 (~)

```
git branch -f main C6
git branch -f bugFix C0
git checkout C1
```

2.4 Reversing Changes in Git

```
git reset local~1
git checkout pushed
git revert pushed
```

3.1 Cherry-pick Intro

```
git cherry-pick C3 C4 C7
```

3.2 Interactive Rebase Intro

```
git rebase -i master~4 --aboveAll
//Remove C2, move C5 up
```

4.1 Grabbing Just 1 Commit

```
git checkout main
git cherry-pick C4
```

4.2 Juggling Commits

```
git rebase -i caption~2 --aboveAll
git commit --amend
git rebase -i caption~2 --aboveAll
git branch -f main caption
```

4.3 Juggling Commits #2

```
git checkout main
git cherry-pick C2
git commit --amend
git cherry-pick C3
```

4.4 Git Tags

```
git tag v0 C1
git tag v1 C2
git checkout C2
```

4.5 Git Describe

```
git commit
```

5.1 Rebasing over 9000 times

```
git rebase master bugFix
git rebase bugFix side
git rebase side another
git rebase another master
```

5.2 Multiple Parents

```
git branch bugWork master~^2~
```

5.3 Branch Spaghetti

```
git checkout one
git cherry-pick C4 C3 C2
git checkout two
git cherry-pick C5 C4 C3 C2
git branch -f three C2
```

How to go from the untracked files to the last commit in the git system and undo the code i written which is untracked by the git system after my last commit

To go from untracked files to the last commit in Git and undo the code you have written, you can follow these steps:

1. Use the command `git status` to see the status of your repository. This will show you a list of all untracked files and changes that have not been committed.
2. Use the command `git add <file>` to add any untracked files to the staging area. Replace `<file>` with the name of the file you want to add. You can also use `git add .` to add all untracked files.
3. Once you have added the files, use the command `git commit -m "Undo changes"` to create a new commit that undoes the changes you made. Replace `"Undo changes"` with a message describing the undo commit.
4. Use the command `git log` to see a list of all commits in your repository. Find the commit that you want to revert to and copy its hash.
5. Use the command `git reset --hard <commit hash>` to revert to the commit you identified in step 4. Replace `<commit hash>` with the hash of the commit you want to revert to.
6. After reverting to the commit, use the command `git status` again to see the updated status of your repository. You should now see that your untracked files are gone and your code has been reverted to the state it was in at the commit you reverted to.

Note: Be careful when using `git reset --hard` as it will remove all changes in your working directory that are not committed or staged.

How to know that your local repository is connected to the remote repository in the git system.

To check if your local Git repository is connected to a remote repository, you can use the command `git remote -v`. This command will list all the remote repositories that your local repository is aware of, along with their URLs.

If your local repository is connected to a remote repository, you should see the name of the remote repository and its URL listed in the output of this command. The name of the remote repository is usually "origin" by default.

If you do not see any remote repositories listed or the output is empty, then your local repository is not connected to any remote repository. In this case, you can use the command `git remote add <name> <remote-url>` to add a new remote repository to your local repository. Replace `<name>` with a name of your choice for the remote repository, and `<remote-url>` with the URL of the remote repository.

Once you have connected your local repository to a remote repository, you can use the command `git push <remote> <branch>` to push your changes from your local repository to the remote repository. Replace `<remote>` with the name of the remote repository and `<branch>` with the name of the branch you want to push your changes to.

How to open a pr request in github and tell me all about pr systems

Pull Requests (PRs) are a feature of GitHub that allow you to propose changes to a repository and collaborate with others on those changes. To open a Pull Request in GitHub, follow these steps:

1. Create a fork of the repository you want to contribute to. A fork is a copy of the repository that you can make changes to without affecting the original repository.
2. Clone your forked repository to your local machine using the `git clone` command.
3. Create a new branch in your local repository to make your changes in. Use the `git checkout -b <branch-name>` command to create and switch to the new branch.
4. Make your changes to the files in your local repository.
5. Use the `git add` command to stage your changes, and the `git commit` command to commit your changes with a descriptive message.
6. Push your changes to the branch on your forked repository using the `git push origin <branch-name>` command.
7. In your forked repository on GitHub, click the "New pull request" button. Select the branch you just pushed your changes to as the "compare" branch, and select the branch you want to merge your changes into as the "base" branch.
8. Write a descriptive title and message for your Pull Request, explaining the changes you have made and why they are necessary.
9. Submit your Pull Request by clicking the "Create pull request" button.

Now, the repository owner or maintainers can review your changes and discuss them with you before deciding whether to merge them into the main repository.

Pull Requests are an important part of the collaborative development process on GitHub. They allow contributors to suggest changes to a repository, and for the repository maintainers to review and accept or reject those changes. Some key features of PRs include:

- Code review: Pull Requests allow for peer review of code changes, allowing contributors to get feedback and ensure that their changes are high quality before being merged into the main branch.
- Collaboration: PRs are a way for contributors to collaborate on code changes, discussing and iterating on changes together.
- Tracking changes: PRs allow you to track changes made to the codebase over time, making it easier to understand why changes were made and who made them.
- Continuous integration: Many organizations use continuous integration tools to automatically build and test code changes submitted via PRs, ensuring that changes don't introduce new bugs or break existing functionality.

Overall, Pull Requests are a powerful tool for open source collaboration, enabling contributors to propose and refine changes to a codebase and work together to improve software over time.

In git system how to retrieve the previous codes without damaging the recent commits.

Suppose there are 6 commits and all are in main branch and the head is on the 6th commit , how will i retrieve the codes of the 2nd commit without affecting the rest of the commits

To retrieve the codes of the 2nd commit without affecting the rest of the commits, you can use the `git checkout` command and create a new branch for the previous commit. Here are the steps to do it:

1. Find the commit hash of the 2nd commit. You can do this by running `git log` and finding the commit hash for the 2nd commit.
2. Create a new branch for the 2nd commit by running `git checkout -b new_branch_name commit_hash`. Replace "new_branch_name" with a name for your new branch, and "commit_hash" with the hash for the 2nd commit.
3. After creating the new branch, you can switch to it by running `git checkout new_branch_name`. This will switch your working directory to the state of the

2nd commit.

4. Retrieve the codes of the 2nd commit from the working directory and copy them to a separate location.
5. Once you have retrieved the codes, switch back to the main branch by running `git checkout main`.
6. The new branch you created in step 2 will still exist, and you can switch back to it at any time to view the codes of the 2nd commit again. To delete the new branch, run `git branch -d new_branch_name`.

Note that retrieving the codes of a previous commit does not affect any of the subsequent commits in the branch. The `git checkout` command simply switches your working directory to the state of the specified commit, allowing you to retrieve the codes from that point in time.