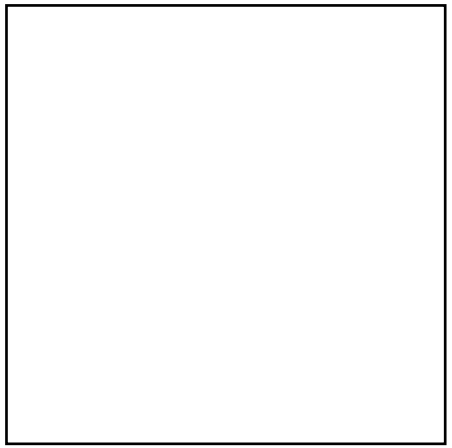
# Fundamental Concepts in

# git!

# Let's start with an empty git repository...

Project Working Directory



Steps 1 and 2 created the "working directory" which is named git-demo.

You will also see this directory referred to as the **working tree**'s root in git.

The .git Repository

```
Follow along with the following commands.
      (You'll type the green text.)
1. learncli$ mkdir git-demo
   # Makes a directory named git-demo
2. learncli$ cd git-demo
   # Changes your shell's directory to it
3. learncli$ git init
   # Initializes a new git repository
4. learncli$ git config user.email "<your email>"
5. learncli$ git config user.name "<name>"
```

Steps 3-5 initialized the hidden .git directory to be an empty Git repository. We'll learn how to configure a system so that steps 4 and 5 are globally defined.

# Now let's add a C file to working directory...

Project Working Directory


The .git Repository

```
learncli$ vim hello.c

vim: Change to Insert Mode (i)

   #include <stdio.h>
   int main()
   {
      printf("hello, world\n");
   }


vim: Change to Normal Mode (Ctrl+[)
save and quit: ZZ
```

# Now let's make an *initial commit...*

Project Working Directory

The .git Repository

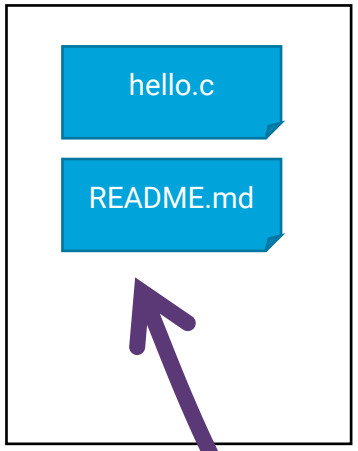
```
learncli$ git add hello.c
# Add the file hello.c to the commit staging area
learncli$ git commit -m "First commit!"
# Form a commit with the given message
```

The "commit staging area" will be discussed soon.

For now, know that a commit (think: project snapshot) with hello.c's contents was created and is stored in the .git directory.

# Let's add a README.md file to the project...

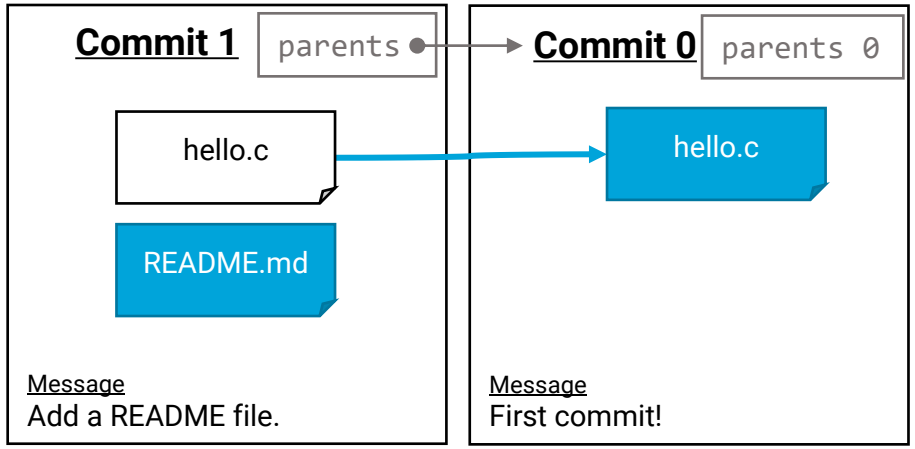
Project Working Directory




```
learncli$ echo "# TODO: Everything" >README.md
# Echo prints its arguments, > redirects output to README.md
learncli$ git add README.md
# Add README.md to the commit staging area
learncli$ git commit -m "Add a README file."
```

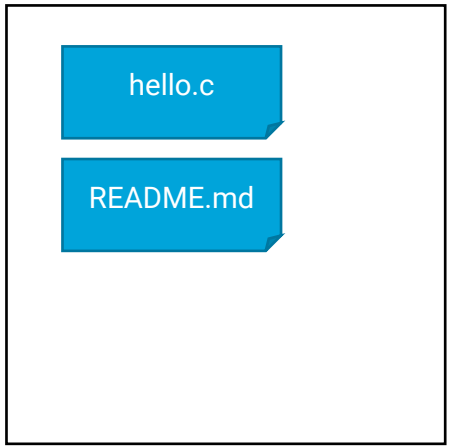
**README.md** is a markdown file documenting the project. When you work on a project it's a *best practice* to have a README file describing how the project is organized, how to build the project, how to contribute to it, and so on.

We'll discuss the updated commit history in the next few slides!



# `git` keeps history of commits made in a repository

## Project Working Directory



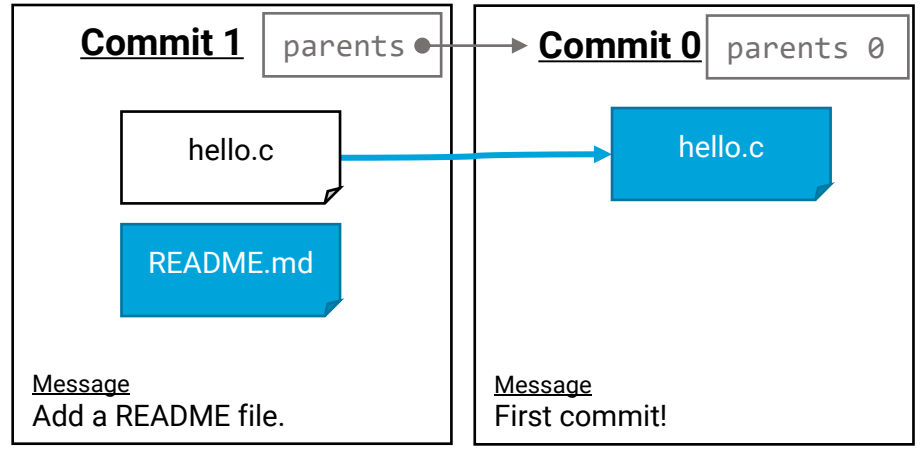

## The .git Repository

This is your project's commit history!

A commit is a **snapshot**, or backup, of your project's files at a specific moment in your project's history.

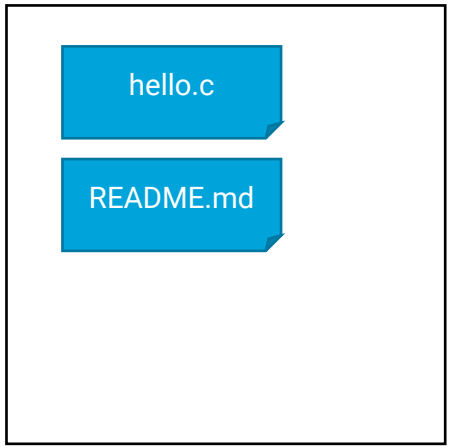
Each of these commits was created when you ran **git commit**. Next we'll discuss exactly what happens as these commits are formed.

Where is this data stored? In the **.git/** folder in your project.



# Commits are linked to parent commits or "previous versions"

## Project Working Directory



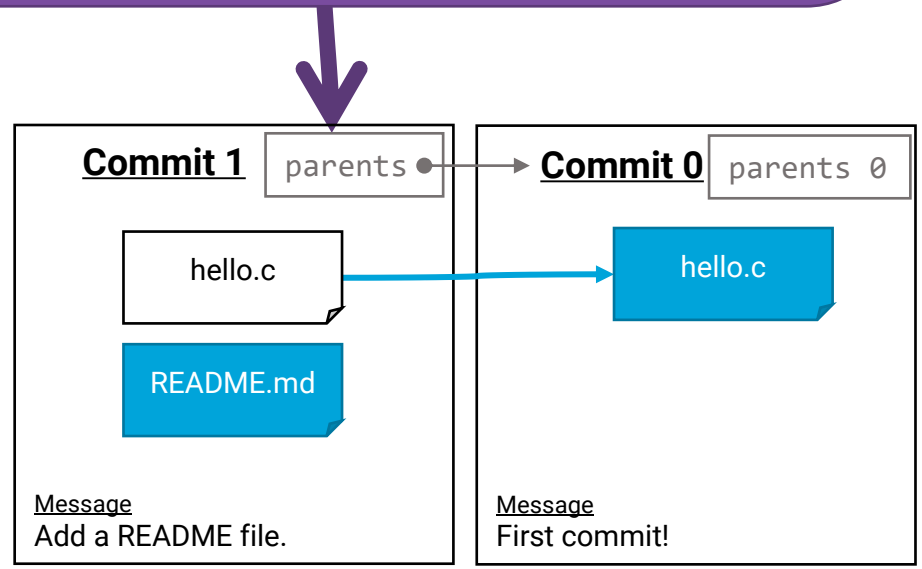

## The .git Repository

The second commit holds a reference to its **parents**, or "previous" commits.

In a simple repository, think of the chain of commits as a singly linked list.

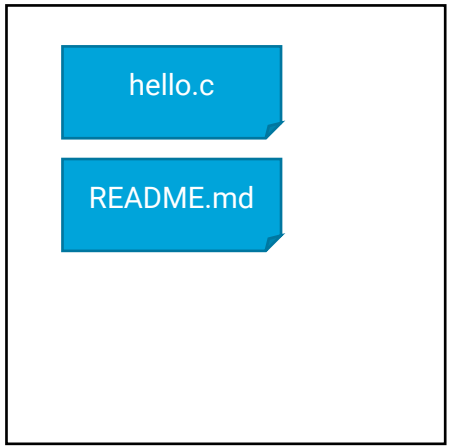
In actuality, the history of commits is a directed-acyclic **graph**.

This distinction is not important until making use of branching and collaboration.



# A commit contains only *important file changes.*

Project Working Directory

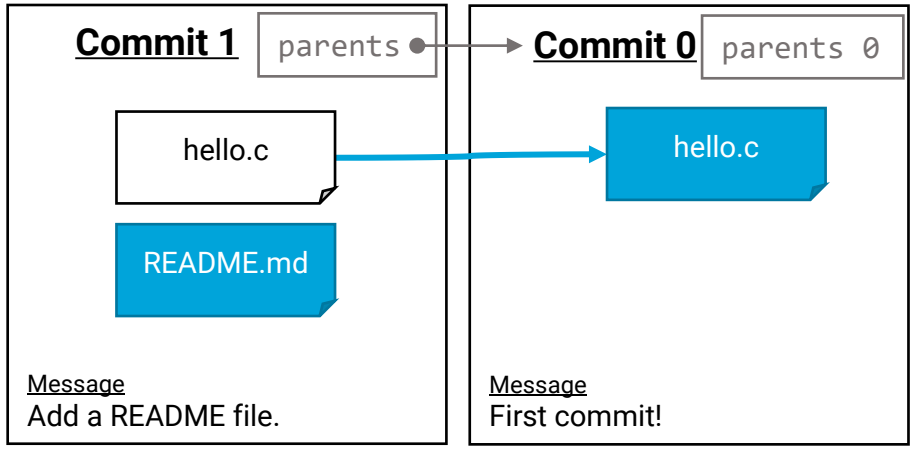



The .git Repository

Only README was added to Commit #1. No changes to hello.c.

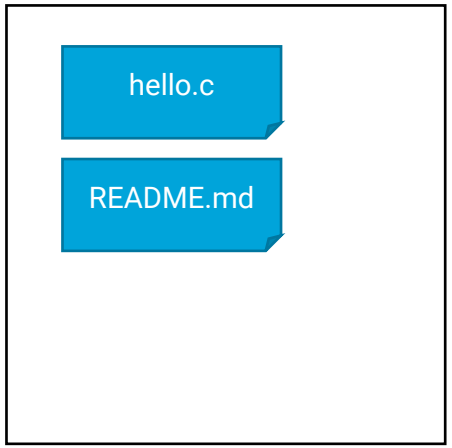
Files whose contents changed in a commit are shown in blue below. Files that did not change are shown in white and *linked* to their last changed version.

What is an **important *file change***?
The person making the commit decides! **It's any file `git add`'ed to the commit.**



# Each commit has a message and other metadata.

Project Working Directory

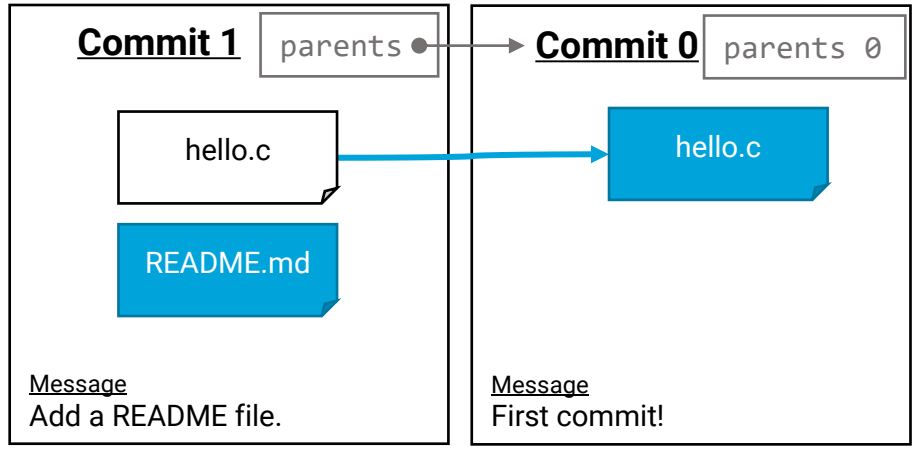



The .git Repository

Each commit has a **message** describing what is important about it.
A message can be a single line, or a header line and narrative.

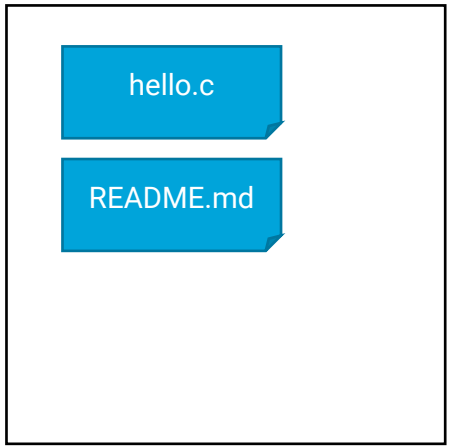
The author of a commit writes this message.
***Your commits should always have informative messages!***

Each commit also has a timestamp and author name/e-mail.



# Each commit has an ID (Identifier)

## Project Working Directory




## The .git Repository

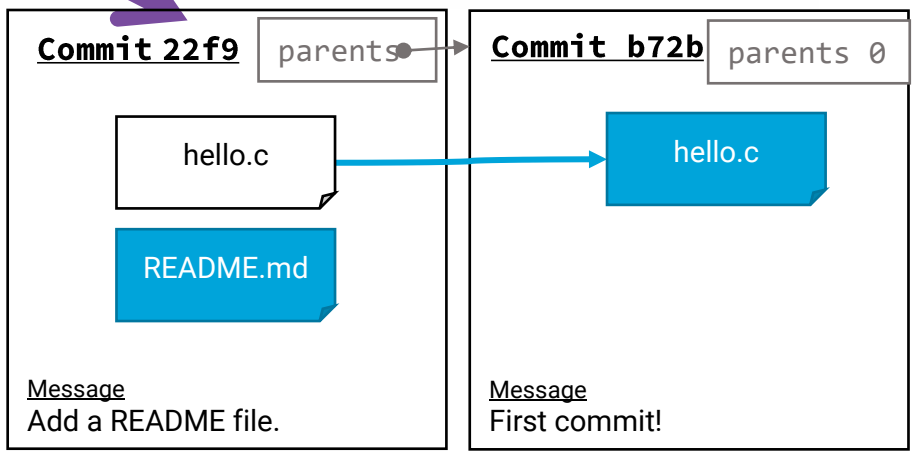
Each commit has a unique identifier (**ID**) formed by ***hashing*** the commit's:
- new/changed file contents of the commit.
- metadata such as parent commits, message, author, timestamp, etc.

The hash algorithm is the cryptographic SHA1 algorithm. It produces IDs like:
`47c2660dcded7b2a27d7b56b017b23574b6200c2`

Referencing commit IDs in git requires only enough characters to uniquely identify a commit ID in a repository's history. **4-chars will be enough for now!**

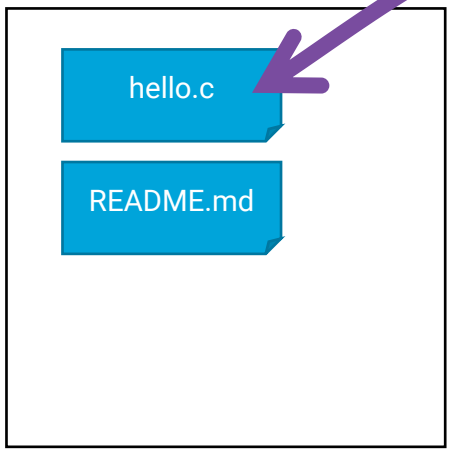
Aside: Hashing is a COMP410/210 subject. If this is new to you, a hand-waving explanation is its an algorithm for generating a unique, fixed-length output from an arbitrary-length input.

The same input will always give the same output hash. Thus, by using the contents of a commit as the input to a hash and the output as the commit's ID, git can use the ID to guarantee correctness and completeness of the data it has saved in the commit.



# What's the big deal?

Project Working Directory



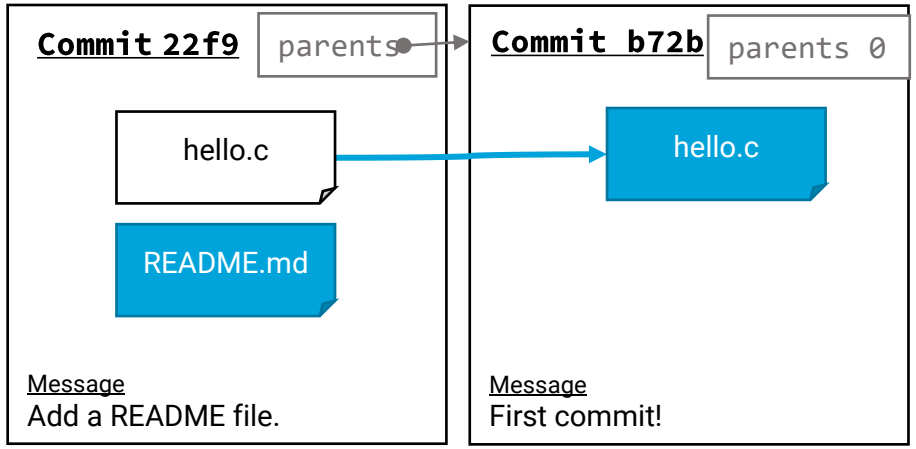

The .git Repository

Suppose you deleted a lot of code in hello.c, saved, and then realized you needed it back...

...with your history in git, it's easy to checkout a committed version of a file without fear of loss.

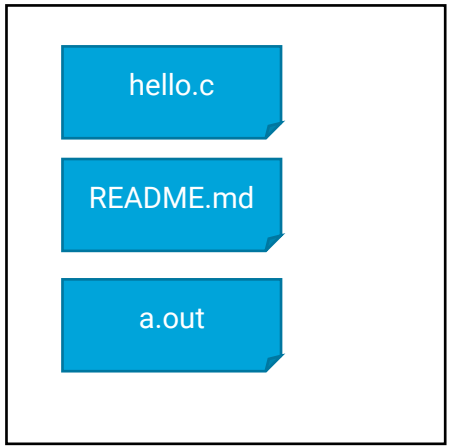
You can also restore all files in a project back to a specific commit in its history.

```
learncli$ git checkout -- hello.c
# Reset hello.c to the last commit's content
```



# How do you get *your* changes into a git repository?

Project Working Directory

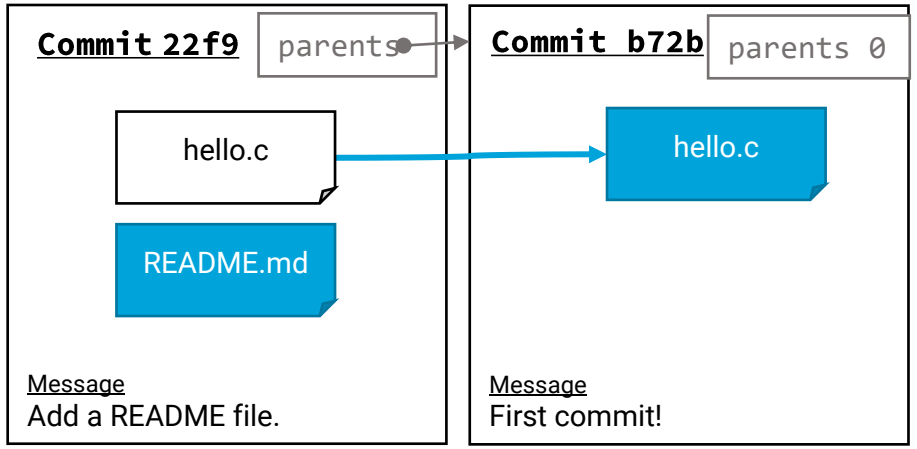



The .git Repository

Imagine you've made changes to hello.c, compiled, and and are ready to commit those changes to the history of the project.

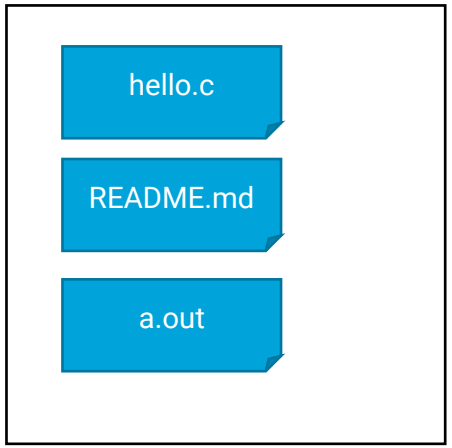
How do you make a commit?

```
learncli$ vim hello.c
vim:
   /hello[press enter]
   o printf("!!!"); Ctrl+[ ZZ
learncli$ gcc -Wall hello.c
learncli$ ./a.out
```



# You craft your next commit in the Staging Area

Project Working Directory




Staging Area

The .git Repository

```
git status
```

You can check the current state of your Staging Area with the **git status** command.

When **`git status`** responds with:

no changes added to commit

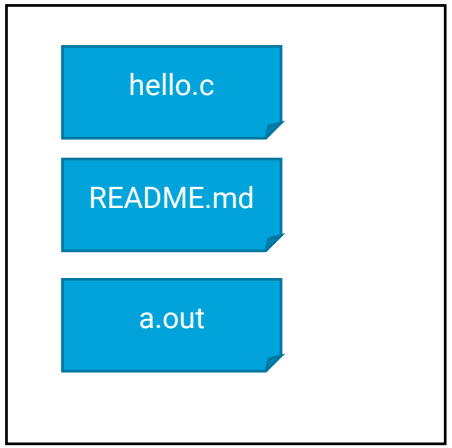
There is nothing in the staging area ready to commit.


The first step in making a commit is **adding** your important file changes to the **staging area**.

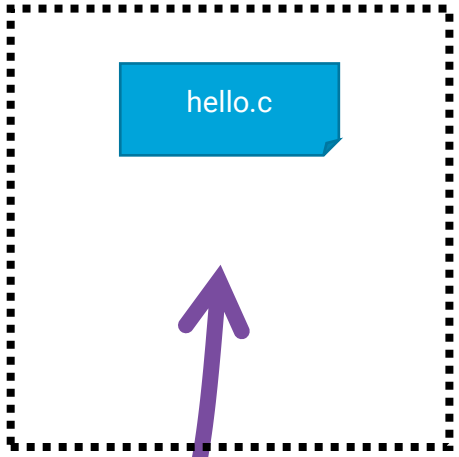
What you add to your staging area is saved behind the scenes in the .git repository.

# Adding files to the Staging Area...

Project Working Directory




Staging Area




The .git Repository

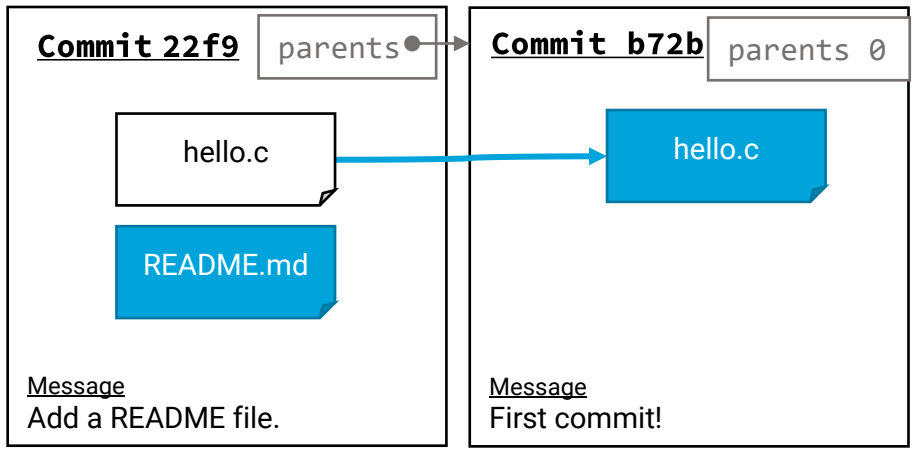
```
git add hello.c
```

```
git status
```

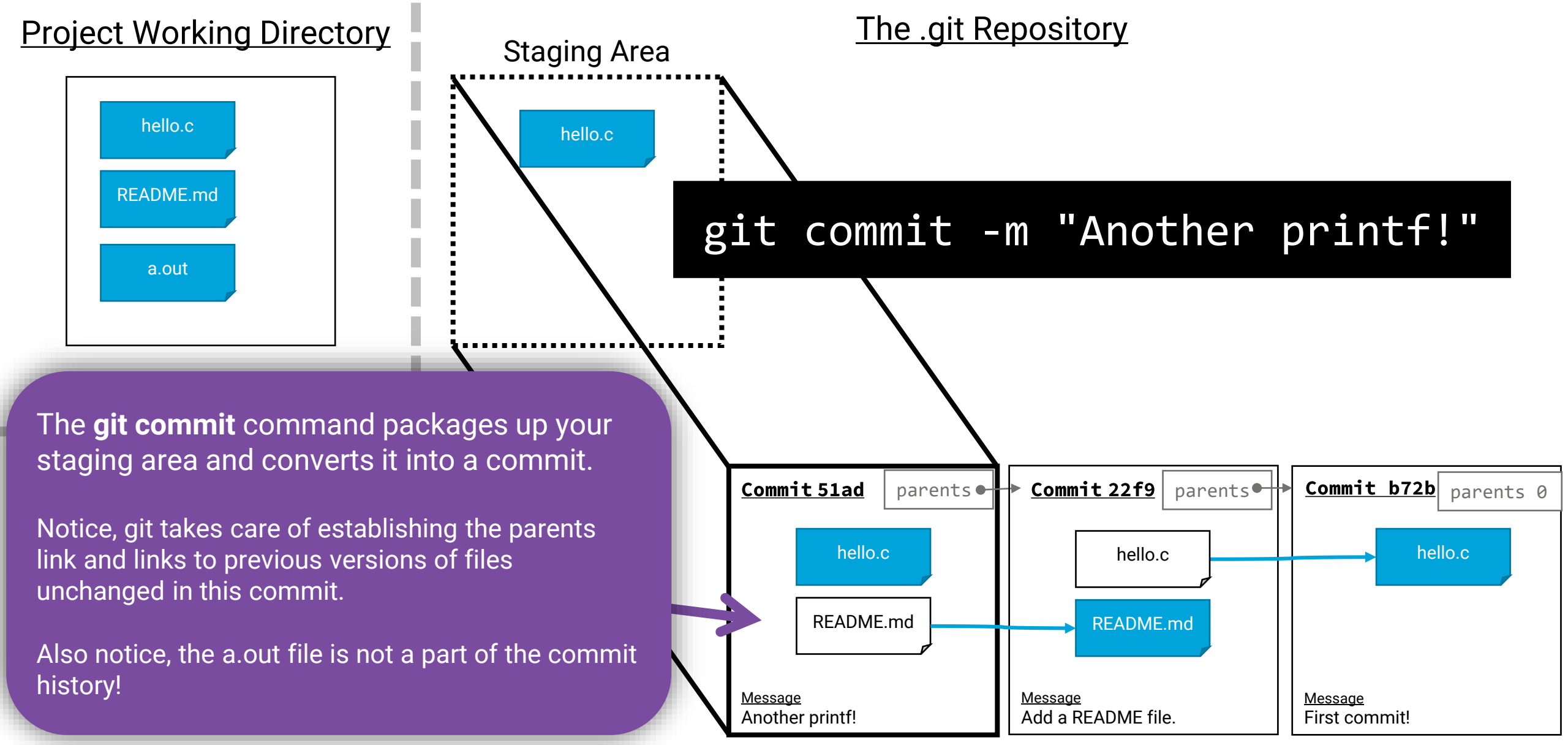
The only important file we want to stage is hello.c, so **we use the git add command (above) to add it to the staging area**.

We specifically *don't want* the *compiled binary file* **a.out** added to our ***source code repository***.

As a general rule of thumb, it is a best practice *not* to store files *built* from source code in version control.



# Make a commit!



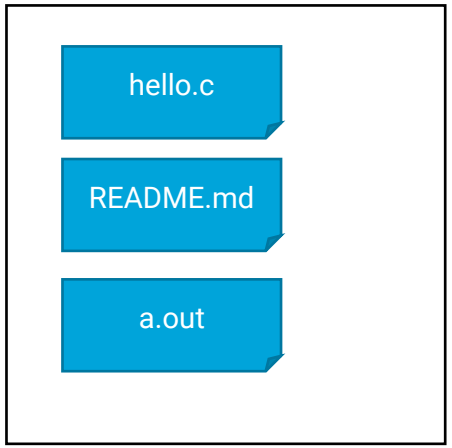

The **git commit** command packages up your staging area and converts it into a commit.

Notice, git takes care of establishing the parents link and links to previous versions of files unchanged in this commit.

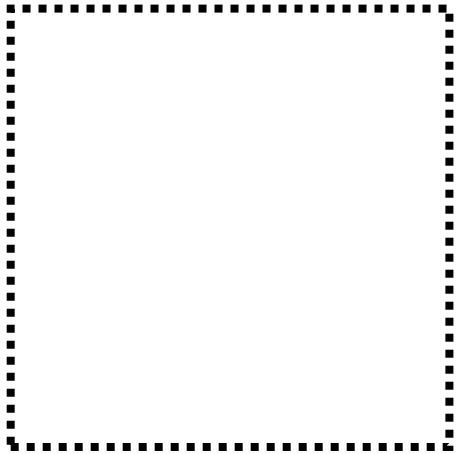
Also notice, the a.out file is not a part of the commit history!

# After a commit, your staging area is clear.

Project Working Directory

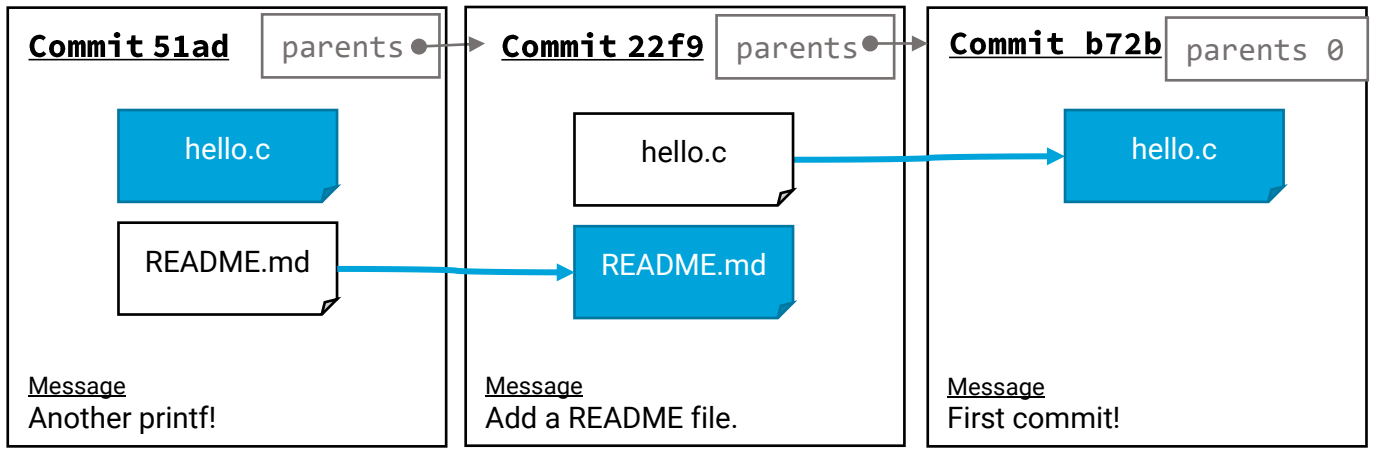



Staging Area



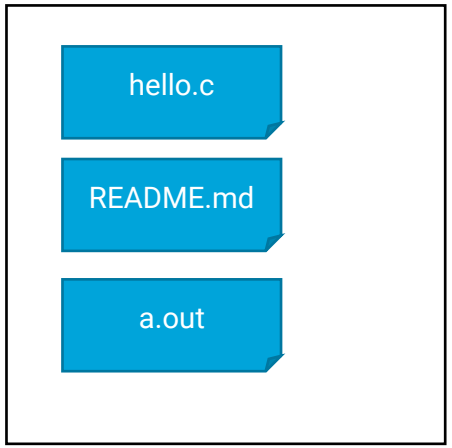
The .git Repository

```
git status
```



# A *branch* is special *named reference* to a commit.

Project Working Directory

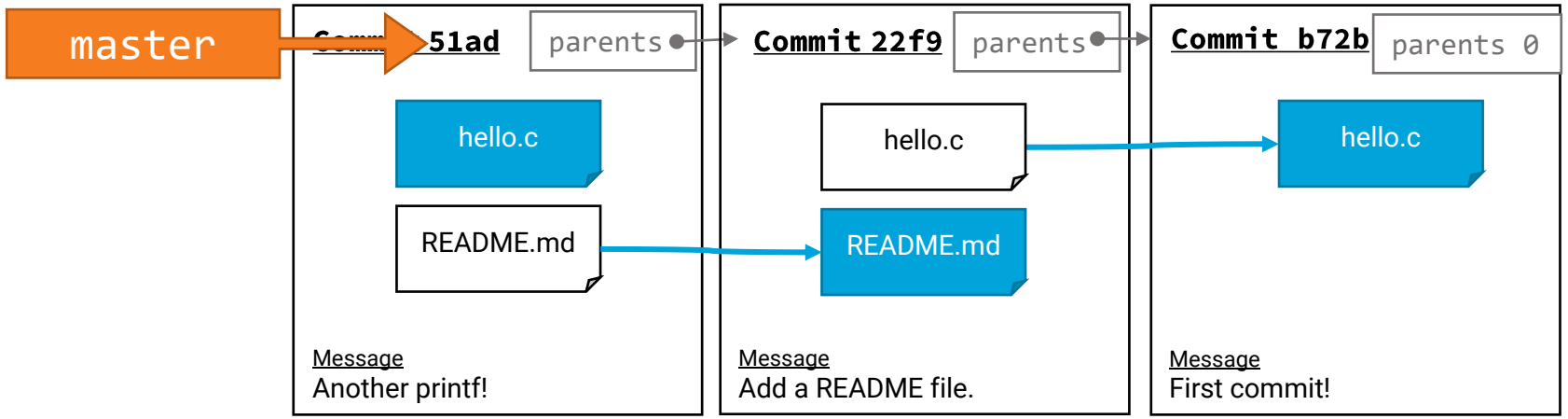



Staging Area

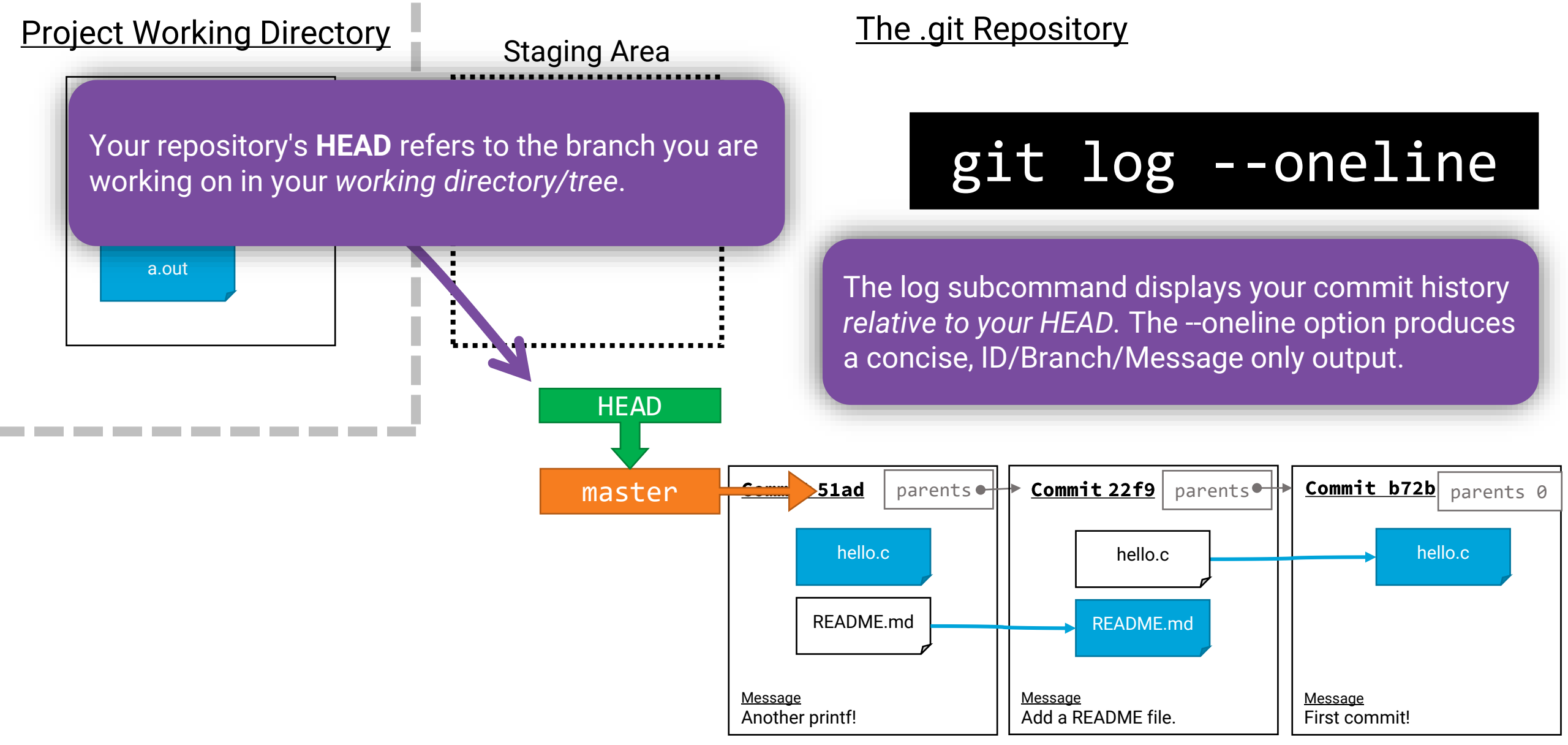
The .git Repository

The default, primary branch is named **master** in git.

A branch is *just* a special reference *to* a specific commit ID in the repository.



# **HEAD** refers to the *current branch* you are working on



# Want to try out an idea? Create and checkout a branch!

Working on a branch is recommended for trying out new ideas.

When a new branch is created it refers to the commit ID you are currently working on.

Thus, two or more branches can refer to the exact same commit ID and commit history.

When you **checkout** a branch:

1. The contents of all files in the commit the branch refers to are copied into your working tree.
2. HEAD is updated to reference the checked-out branch

## The .git Repository

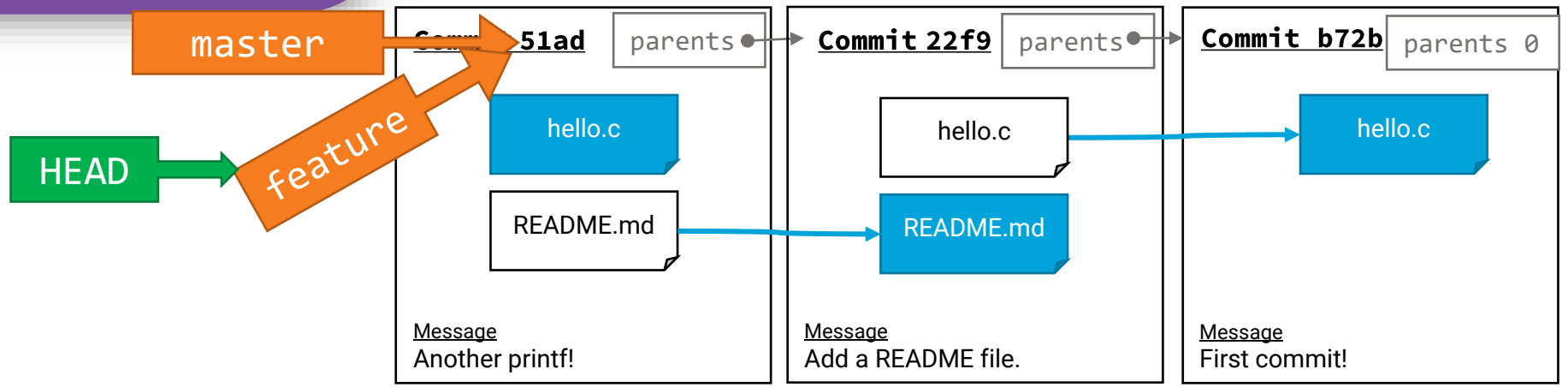
```
git branch # list branches
```

```
git branch <name> # create branch named <name>
```

```
git checkout <name> # checkout branch <name>
```

The two above commands are usually done together in a single command:

```
git checkout -b <name>
# create and checkout branch <name>
```

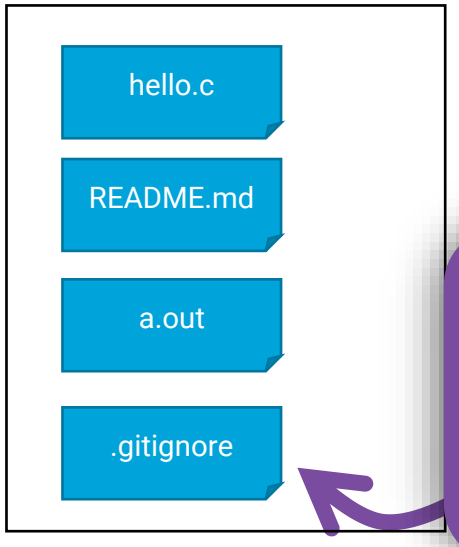


# Making changes before committing...

Project Working Directory

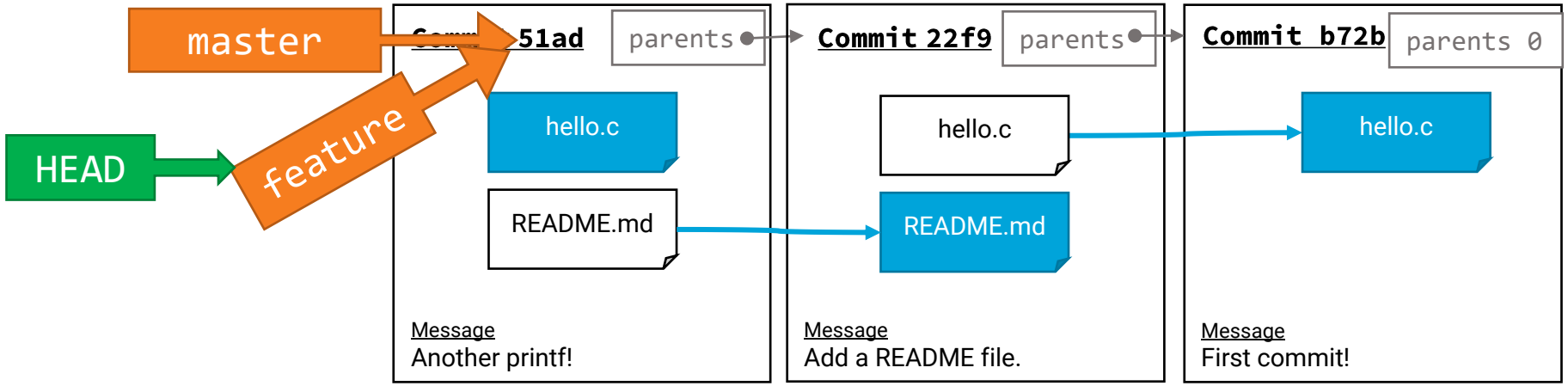
Staging Area

The .git Repository

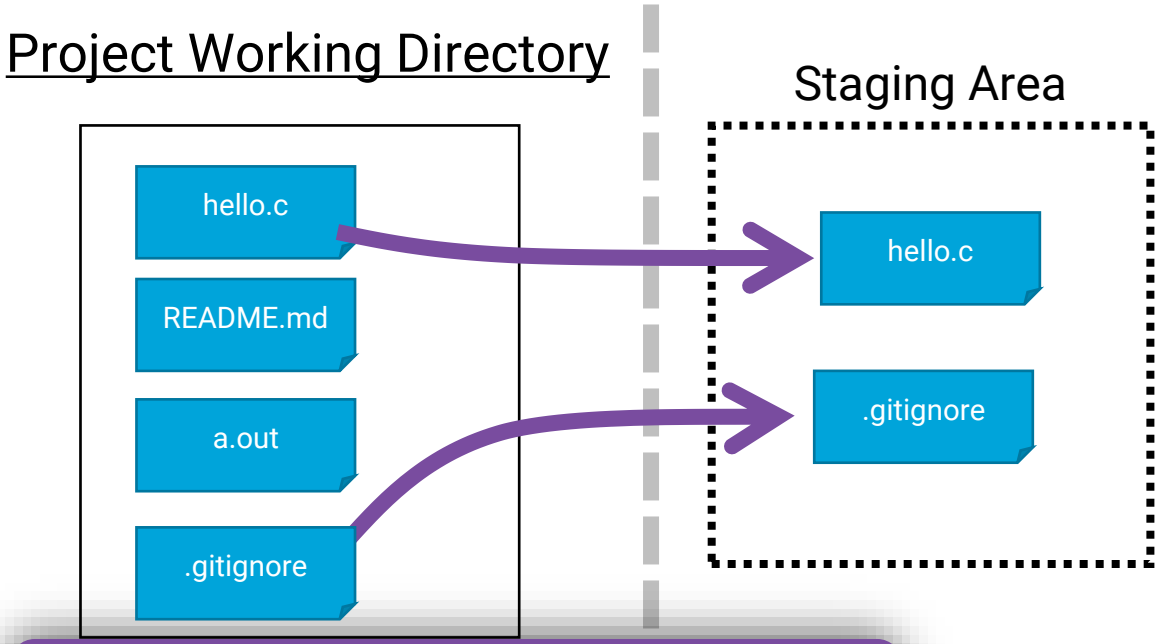



Any filenames listed in the a **.gitignore** file are ignored by `git status` and `git add`.

```
learncli$ echo "a.out" >>.gitignore
learncli$ cat .gitignore
learncli$ vim hello.c
vim:
   /hello[press enter]
   o printf("wow"); Ctrl+[ ZZ
learncli$ gcc -Wall hello.c
learncli$ ./a.out
```



# Adding files to staging before committing...

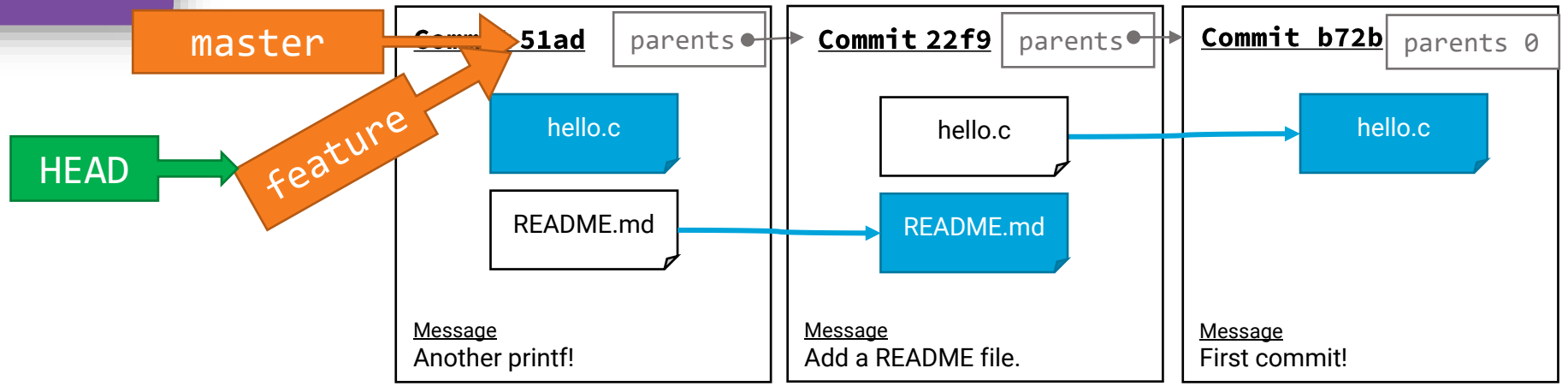



The .git Repository

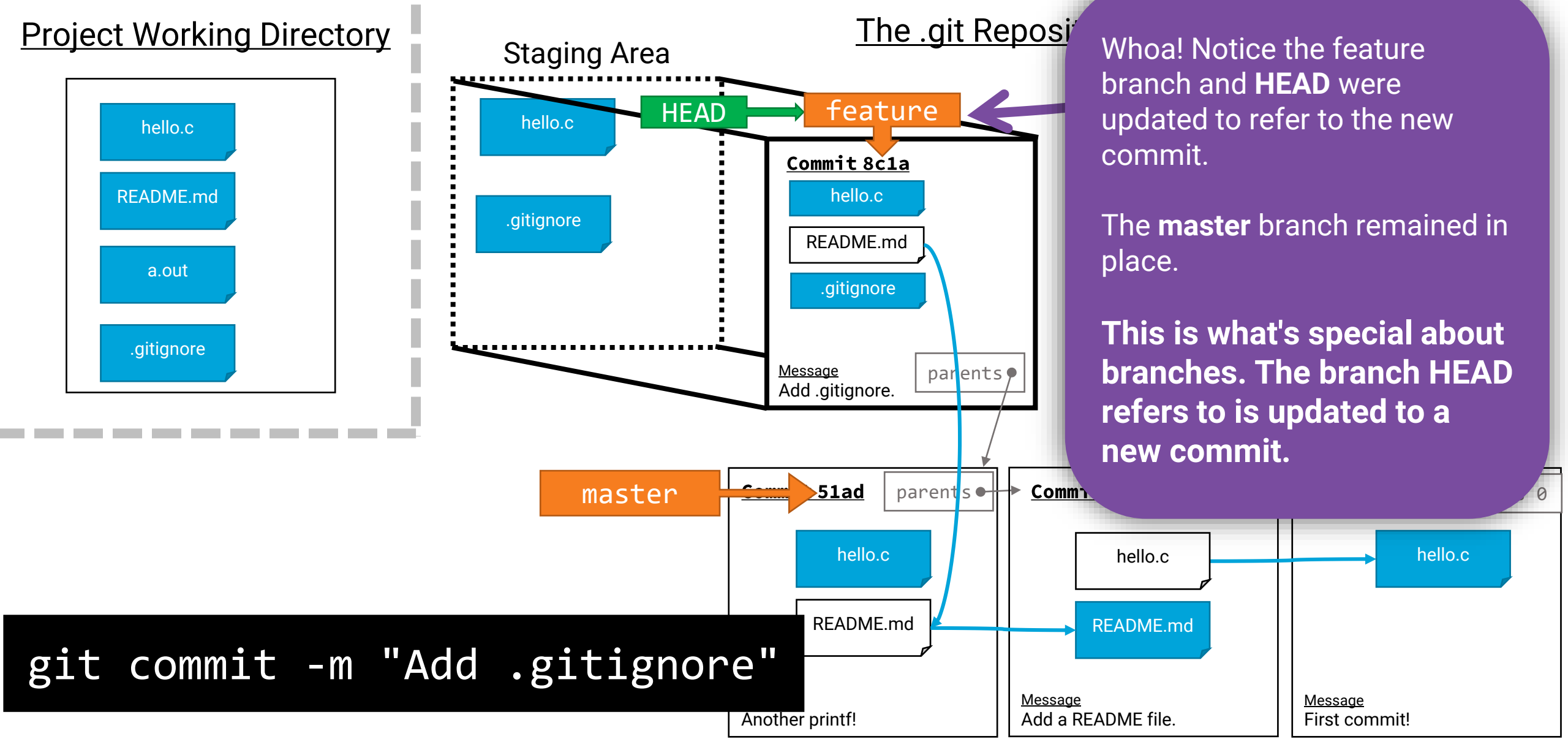
```
git add hello.c
```

```
git add .gitignore
```

Step 1) **git add** to staging area.



# Making a commit updates the branch HEAD refers to



# What's the big idea of branches?

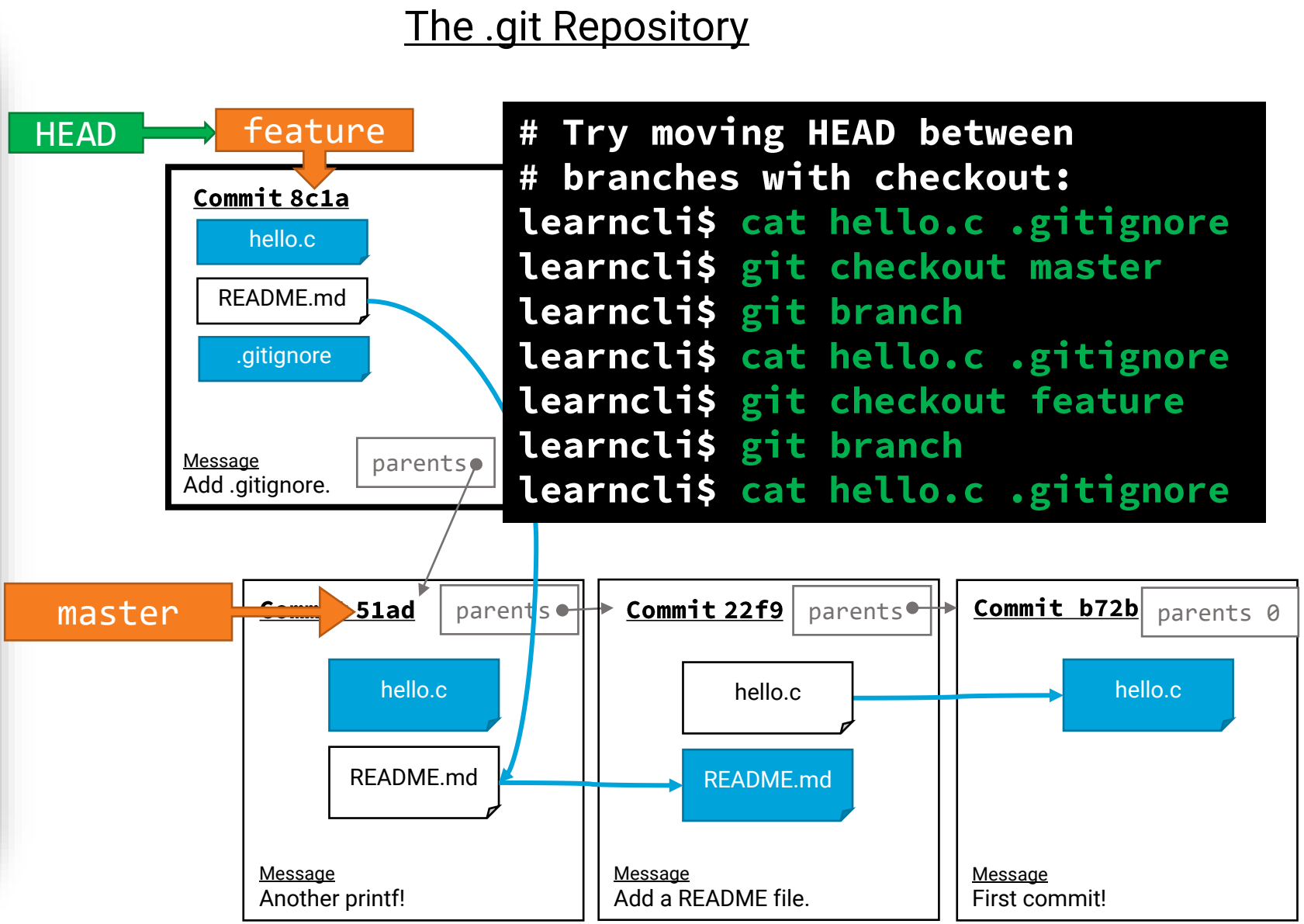
Imagine making additional commits on this "feature branch".

You can *easily* checkout the master branch and continue working from there, though.

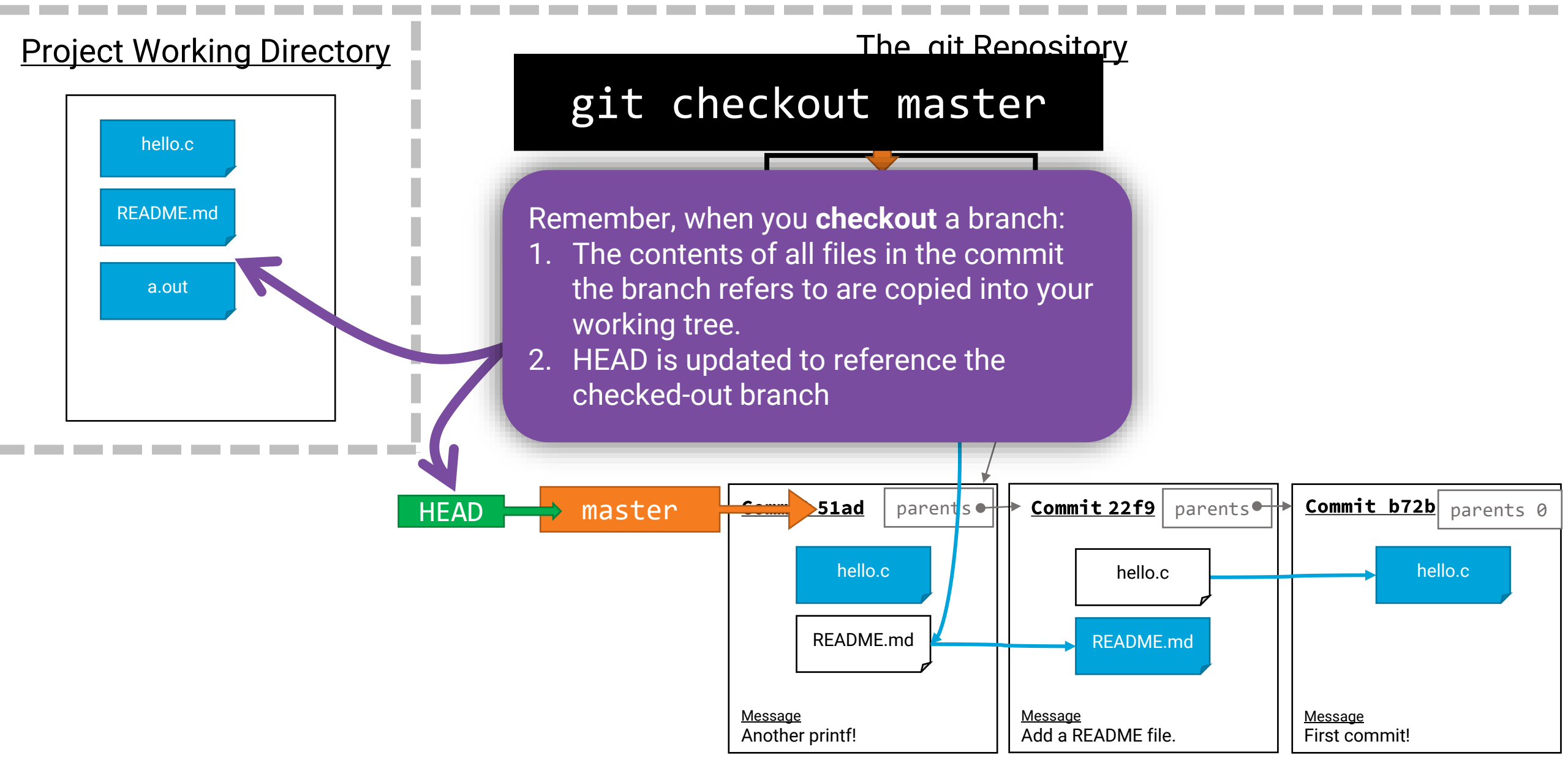
You could also easily start a *other branches* to explore other feature ideas.

**Good idea?** Merge the feature branch back into the master branch. (Up next!)

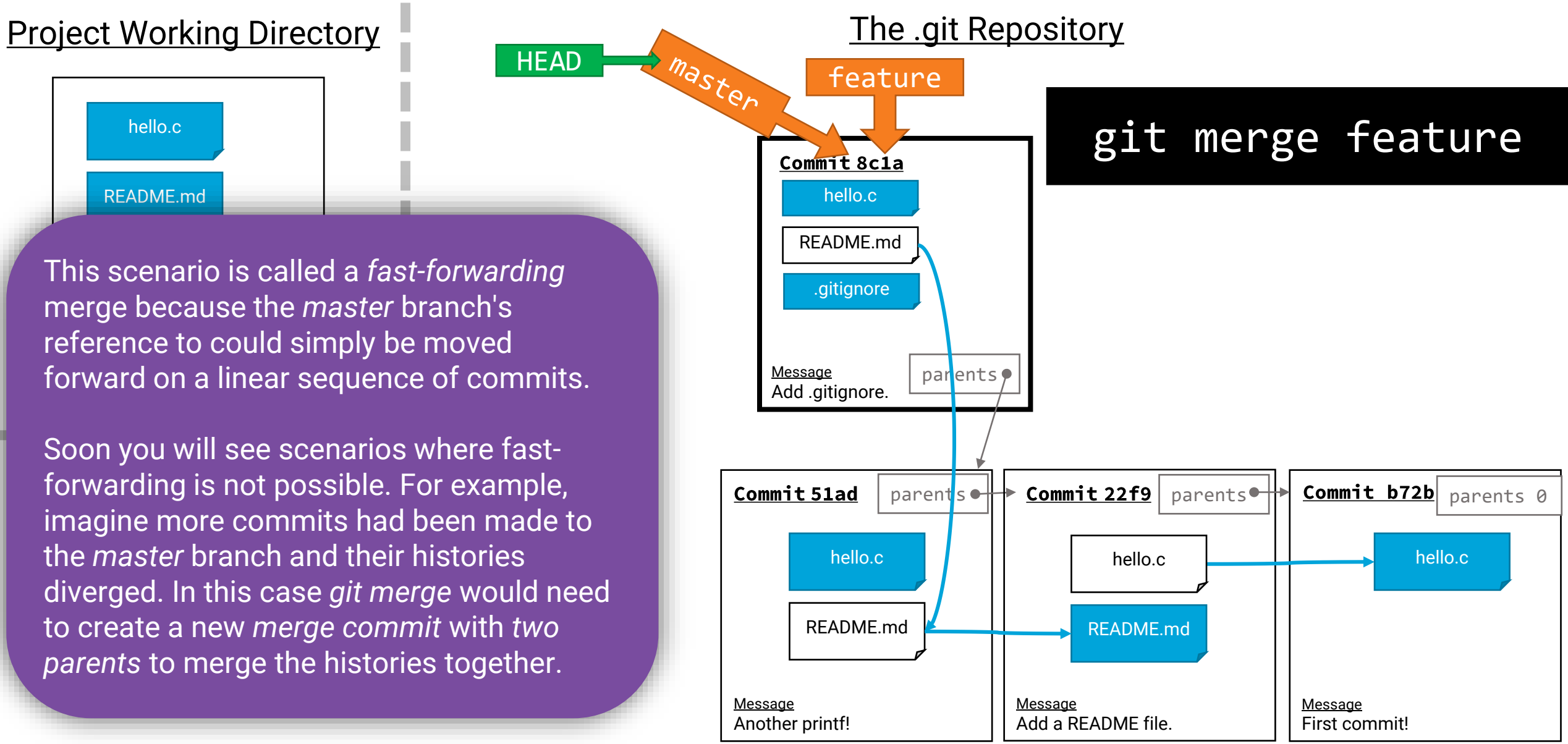
**Bad idea?** Checkout the master branch and delete the feature branch.



# To *merge branches*, first checkout the branch you're merging onto.



# Then, issue the `git merge` command

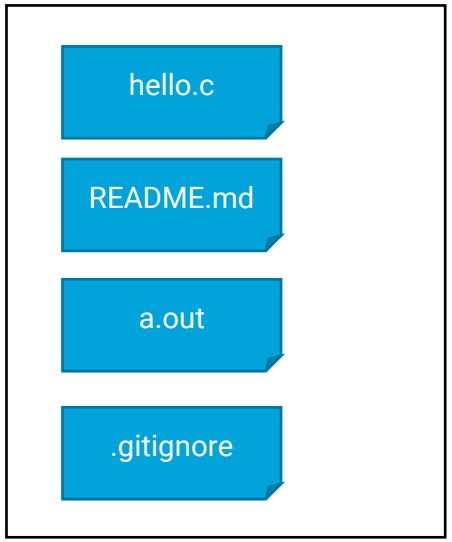



This scenario is called a *fast-forwarding* merge because the *master* branch's reference to could simply be moved forward on a linear sequence of commits.

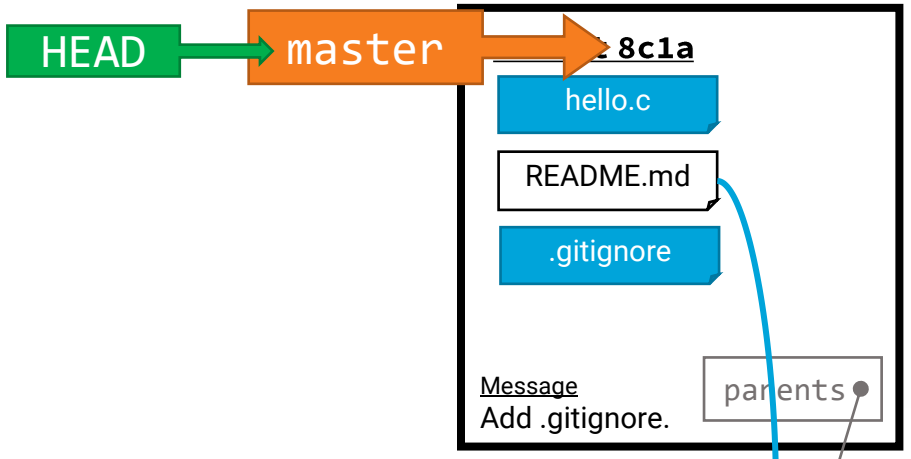
Soon you will see scenarios where fast-forwarding is not possible. For example, imagine more commits had been made to the *master* branch and their histories diverged. In this case *git merge* would need to create a new *merge commit* with *two parents* to merge the histories together.

# Deleting branches you no longer need...

## Project Working Directory




## The .git Repository

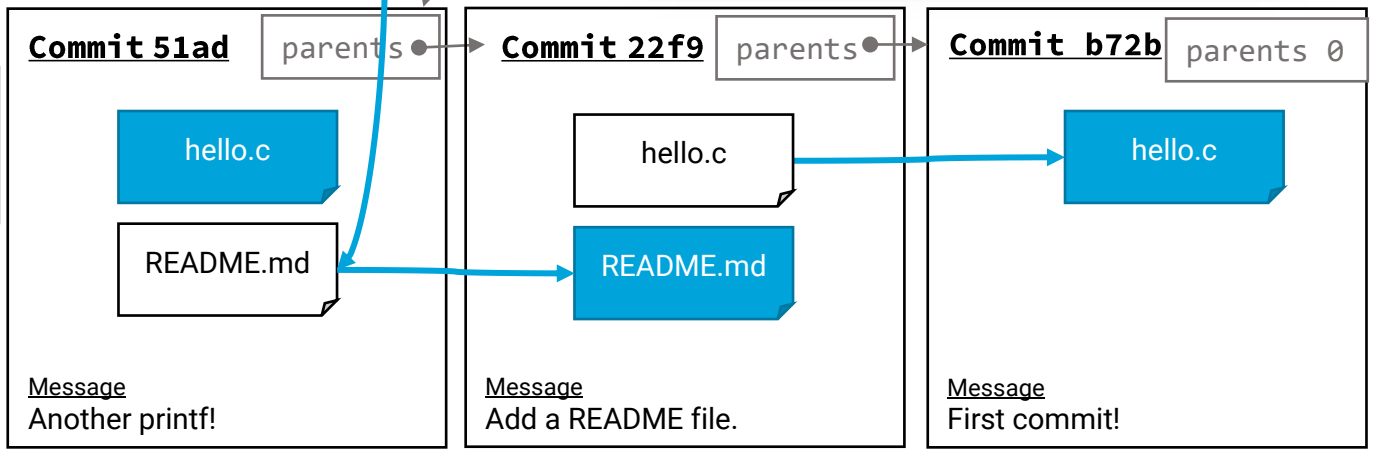



In general, git is designed to make it difficult for you to accidentally destroy work.

If you had tried to delete the feature branch before merging it would have told you it contained unmerged changes and asked if you were really sure.

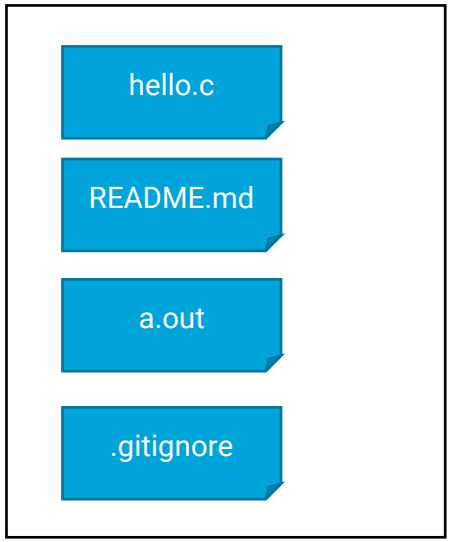
Additionally, when you checkout a branch, if you have uncommitted changes in your working directory it will prevent you from doing so and overwriting them.

```
git branch --delete feature
```

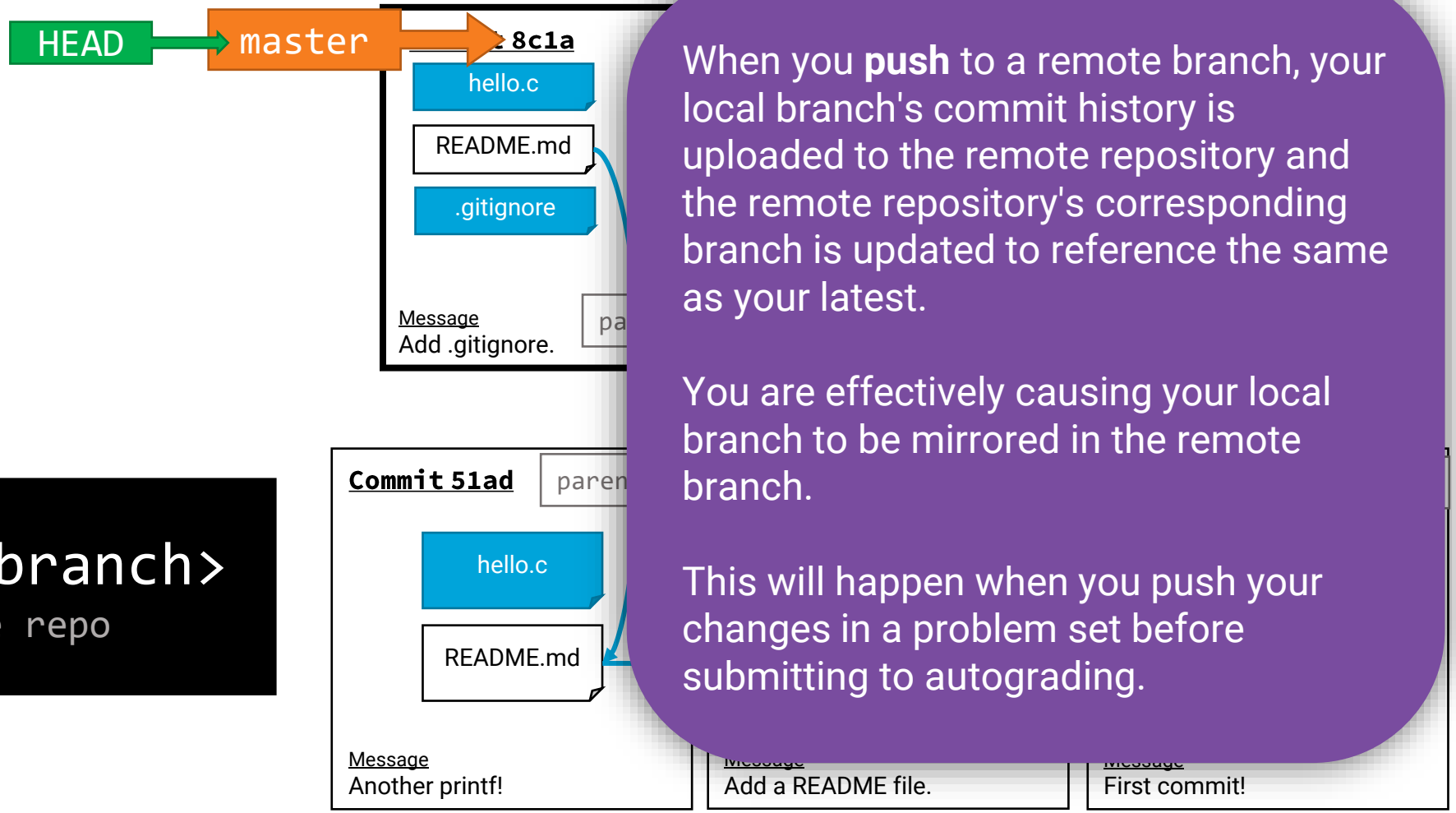


# Pushing Changes to a *Remote Repository*

## Project Working Directory




## The .git Repository




When you **push** to a remote branch, your local branch's commit history is uploaded to the remote repository and the remote repository's corresponding branch is updated to reference the same as your latest.

You are effectively causing your local branch to be mirrored in the remote branch.

This will happen when you push your changes in a problem set before submitting to autograding.

```
git push origin <branch>
# Uploads changes to remote repo
```