Systems Programming & Engineering

COS 316: Principles of Computer System Design

Lecture 3

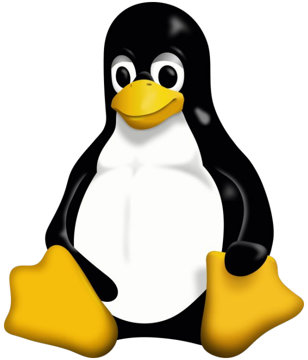
Leon Schuermann

Agenda

1. Challenges in Real-World Systems Engineering
2. Version Control Systems
3. Continuous Integration

Systems Engineering Challenges

- Designing and building systems in large real-world projects is hard
  - Vast set of requirements
  - Constrained environments and resources
  - Complex layering models and abstractions
- Prominent example: the Linux kernel
  - (Amongst the) largest single software project(s)
  - > 27 Mio lines of code[1], developed in the open since 1991
  - In 2019, 74k individual changes contributed by > 4k individual contributors
  - Supports incredibly wide range of hardware (routers, mobile phones, workstations, servers,...)
  - Extensive range of application interfaces



[1]: https://www.phoronix.com/news/Linux-Git-Stats-EOY2019

Even changes in smaller projects can be complex…

Tock 2.0: implement Callback swapping restrictions (v3) #2462

Edit | <> Code

Closed **lschuermann** wants to merge 24 commits into `master` from `tock2-callback-swap-prevention`

Conversation 81 | Commits 24 | Checks 0 | Files changed 114 | +1,943 −674

lschuermann commented on Mar 3, 2021 • edited by hudson-ayers — Member

Pull Request Overview

This pull request is a follow-up to #2282 and #2445. This was done jointly together with **@hudson-ayers**. Thanks Hudson!

Issue

Essentially, the protections introduced in this PR are required because in the Tock 2.0 system call architecture, with every call to `subscribe`, the previous `Callback` provided by the application must be returned. If we were to not enforce any constraints here, this would have serious side effects:

- a `Callback` subscribed by process A could be handed back to process B.

  This leaks both the callback pointer and the `appdata` field of the `Callback` to another application.

- a `Callback` from one capsule could be given to another capsule and returned as part of a call to subscribe there.
- a `Callback`, passed to a driver under subscribe (subdriver) number `x` could be returned as part of a call to `subscribe` to the same driver under subscribe (subdriver) number `y`, where `x != y`.
- a capsule could pass back a *null callback* (with callback pointer and appdata being 0), where a process has actually subscribed a `Callback` with non-null values before.

  All of these cases could cause inconsistencies and undefined behavior in userspace, if a process were to rely on the fact that the returned `Callback` is the one previously passed to the capsule.

Reviewers
alistair23
hudson-ayers
bradjc

Assignees
alevy

Labels
kernel | P-Significant | tock-2.0 | WG-OpenTitan

Projects
None yet

Milestone
No milestone

Development

bradjc commented on Mar 4, 2021 · Member

1. So this doesn't prevent a capsule from returning the wrong callback, but rather it makes it possible for the core kernel to check that the proper callback was returned?
2. Why is a capsule with two grants problematic? Is Kernel.grant_num_mapping only to prevent multiple grants in one capsule?
3. In my opinion needing the external macro crates is problematic.

lschuermann commented on Mar 4, 2021 • edited · Member · Author

> 1. So this doesn't prevent a capsule from returning the wrong callback, but rather it makes it possible for the core kernel to check that the proper callback was returned?

Yes. We thought about ways to do the entire check statically (e.g. through const generics, etc.) but ultimately couldn't find a way to make that work. The compiler can only work with the type information it has, so pretty much every approach relying on this would generate a bunch of types at compile time, require use of generics and monomorphisation would increase code size (if at all possible).

> 2. Why is a capsule with two grants problematic?

We use the `ProcessCallbackFactory` in the grant initialization, whose purpose is to ensure that per (driver, process) combination, no two Callbacks for the same subdriver number can be created. If there were to exist two Grant regions for a single driver (which would produce a `ProcessCallbackFactory` two times), we can't enforce that invariant anymore, without not also storing the `ProcessCallbackFactory` state per (driver, process) somewhere.

> Is Kernel.grant_num_mapping only to prevent multiple grants in one capsule?

Yes.

> 3. In my opinion needing the external macro crates is problematic.

That's unfortunate, though not an issue. The only crate *absolutely required* to build macros is `proc-macro`, which is built by the Rust compiler team (the compiler essentially "links" against that interface defined there, as procedural macros are extensions to

...

jrvanwhy commented on Mar 8, 2021 Member ···

+1 to Hudson's explanation. My shortened version is that yes we technically could make it the responsibility of the board main file, but in practice it would be easy to make mistakes in the board main file if we do so.

bradjc commented on Mar 11, 2021 Member ···

OK next attempt.

What about having callback store a `Option<DriverNum>`? The default callback created at initialization will have `None`. But, any callback passed in will have `Some(DriverNum)`. If the same <process, driver, subscribe> is called again, then the `DriverNum` has to match. For the first call to <process, driver, subscribe>, the driver number would be None and that would just match.

hudson-ayers commented on Mar 11, 2021 • edited by bradjc Member ···

> OK next attempt.
>
> What about having callback store a `Option<DriverNum>`? The default callback created at initialization will have `None`. But, any callback passed in will have `Some(DriverNum)`. If the same <process, driver, subscribe> is called again, then the `DriverNum` has to match. For the first call to <process, driver, subscribe>, the driver number would be None and that would just match.

That is pretty close to Leon's first approach. The problem is there is no way for the kernel to know the same <process, driver, subscribe> has been called before. Even if it is only possible for trusted kernel code to modify the field containing `Option<DriverNum>`, consider this scenario:

Capsule A and Capsule B each have 2 callbacks (subdriver num 0 and 1 for both). Thus there are 4 callbacks total: A0, A1, B0, B1. These capsules have a reference to each other and cooperate maliciously with the goal of violating the kernel guarantees

...

Systems Engineering Challenges

- How do you develop and maintain a project…
  - that is too large to be developed by a single individual,
  - honoring new feature requests,
  - without breaking any existing users / subsystems,
  - sustainably, over a long period of time,
  - in an auditable way?

- How do systems engineers solve these problems? They build systems, of course!
- Today, we introduce two systems which help with these challenges:

Distributed Version Control
(git)

Continuous Integration
(GitHub Actions)

You will use both types of systems for the programming assignments!

Version Control Systems

- Development rarely goes perfect
  - Introducing new bugs with changes over time
  - Deleting code believed to no longer be useful
  - External requirements change
- Version Control Systems track the state of a project over time

A Student’s Version Control “System”

```
2023-04-05_design_prompt.docx
2023-05-14_design_prompt.docx
2023-05-20_design_prompt.docx
2023-05-20-01_design_prompt.docx
```

```
initial_draft.docx
wip.pdf
final_submission.pdf
final_submission_2.pdf
final_draft_submission.pdf
final-final.pdf
```

https://docs.google.com/presentation/d/1xDJ8crO7nBBIfV9eh5kiAHLse Work

Today, 2:15 PM

Fit

Total: 2 edits

5

6

7

Systems Engineering Challenges

- How do you develop and maintain a project…
  - that is too large to be developed by a single individual,
  - honoring new feature requests,
  - without breaking any existing users / subsystems,
  - sustainably, over a long period of time,
  - in an auditable way?
- How do systems engineers solve these problems? They build systems, of course!
- Today, we introduce two systems which help with these challenges:

Distributed Version Control *(git)*

Continuous Integration *(GitHub Actions)*

You will use both types of systems for the programming assignments!

Version history

All versions

TODAY

September 10, 2:15 PM
Current version
Leon Schuermann

September 10, 2:15 PM
Leon Schuermann

September 10, 1:58 PM
Leon Schuermann

September 10, 1:57 PM
Leon Schuermann

September 10, 1:57 PM
Leon Schuermann

September 10, 1:57 PM

Show changes

Version Control Systems

- These version control schemes are <u>not</u> suitable for software projects
  - Revisions are taken automatically, or at arbitrary points in time
    - Do not track single (“*atomic*”) changes to a given project
  - No semantic information associated with versions
    - Which version was a certain bug introduced in?
    - Does a given version contain a feature / bug?
  - Linear version history
    - No support for separating concurrent work (e.g., by multiple developers)
    - Copies of a project/document cannot be automatically reconciled

Introducing git

- Created by Linus Torvalds (creator of Linux) in 2005
- Designed as a Version Control System (VCS) for the Linux kernel
- Very popular, but not the only VCS
  (Mercurial, SVN, CVS, Perforce, darcs, Pijul, …)

- You will learn how git works across two lectures
  - A practical guide (this lecture)
  - A deep-dive into git’s underlying architecture (09/21, Prof. Levy)

git 101

- Git tracks content in a *git repository*
  - Let's create one now!

```
leons@silicon ~/cos316-l03> mkdir cos316-repo

leons@silicon ~/cos316-l03> cd cos316-repo/

leons@silicon ~/c/cos316-repo> git init
Initialized empty Git repository in /home/leons/cos316-l03/cos316-repo/.git/

leons@silicon ~/c/cos316-repo (main)>
```

git 101

- Git tracks content in a *git repository*
    - Let’s create one now!
- A repository manages a given folder (i.e. your project’s root directory)

```
leons@silicon ~/c/cos316-repo (main)> git status
On branch main

No commits yet

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

leons@silicon ~/c/cos316-repo (main)> echo '
                                   package main

                                   import "fmt"

                                   func main() {
                                       fmt.Println("Hello World!")
                                   }
                                   ' > test.go

leons@silicon ~/c/cos316-repo (main)> go run test.go
Hello World!
```

```
leons@silicon ~/c/cos316-repo (main)> git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    main.go

nothing added to commit but untracked files present (use "git add" to track)
```

git 101

- Git tracks content in a *git repository*
  - Let’s create one now!
- A repository manages a given folder (i.e. your project’s root directory)
- Git tracks versions through *commits*
  - A commit is a snapshot of the repository directory
  - It only includes *changes* marked for inclusion

```
leons@silicon ~/c/cos316-repo (main)> git add main.go

leons@silicon ~/c/cos316-repo (main)> git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
	new file:   main.go
```

git 101

- Git tracks content in a *git repository*
    - Let's create one now!

- A repository manages a given folder (i.e. your project's root directory)

- Git tracks versions through *commits*
    - A commit is a snapshot of the repository directory
    - It only includes *changes* marked for inclusion

- `git add` adds a file to the "*staging area*"
    - The next commit will include whichever changes are *staged*
    - `git add` "freezes" the file version added to the staging area – let's see this in action

```
leons@silicon ~/c/cos316-repo (main)> sed -i 's/World/COS316/g' main.go

leons@silicon ~/c/cos316-repo (main)> cat main.go
...
func main() {
    fmt.Println("Hello COS316!")
}

leons@silicon ~/c/cos316-repo (main)> git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   main.go

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   main.go
```

`main.go` is added *and* modified! The added version still prints "Hello World!", but the current *in-tree* version prints "Hello COS316!"

git 101

- Let's create our first commit!
- `git commit` records a snapshot of the entire repository
  - But only including the changes from the staging area
- Best practice: always check what you're committing!
  - Use `git diff --staged` to view the currently staged changes
  - Use `git diff` to view changes not currently staged

```
leons@silicon ~/c/cos316-repo (main)> git diff --staged
diff --git a/main.go b/main.go
new file mode 100644
index 0000000..b1b14d0
--- /dev/null
+++ b/main.go
@@ -0,0 +1,7 @@
+package main
+
+import "fmt"
+
+func main() {
+	fmt.Println("Hello World!")
+}
```

```
leons@silicon ~/c/cos316-repo (main)> git diff
diff --git a/main.go b/main.go
index b1b14d0..d94cebf 100644
--- a/main.go
+++ b/main.go
@@ -3,5 +3,5 @@ package main
 import "fmt"

 func main() {
-	fmt.Println("Hello World!")
+	fmt.Println("Hello COS316!")
 }
```

git 101

- Let’s create our first commit!
- `git commit` records a snapshot of the entire repository
  - But only including the changes from the staging area
- Best practice: always check what you’re committing!
  - Use `git diff --staged` to view the currently staged changes
  - Use `git diff` to view changes not currently staged
- Looking good? Use `git commit` to finalize your commit!
  - Record some semantic information with this change:
    `git commit -m “This is a commit message”`
  - Writing good commit messages is it’s own science…

```
leons@silicon ~/c/cos316-repo (main)> git commit -m "Add Hello World
application"
[main (root-commit) fa93736] Add Hello World application
 1 file changed, 7 insertions(+)
 create mode 100644 main.go

leons@silicon ~/c/cos316-repo (main)> git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   main.go

no changes added to commit (use "git add" and/or "git commit -a")

leons@silicon ~/c/cos316-repo (main)> git show
commit fa937364b216ce78a07356b096618fbf85eca523 (HEAD -> main)
Author: Leon Schuermann <leon@is.currently.online>
Date:   Sun Sep 10 18:17:37 2023 -0400

      Add Hello World application

diff --git a/main.go b/main.go
...
```

git 101

- Commits are identified by a 40-character “commit id”

```
[main (root-commit) fa937364] Add Hello World application

leons@silicon ~/c/cos316-repo (main)> git show
commit fa937364b216ce78a07356b096618fbf85eca523 (HEAD -> main)
Author: Leon Schuermann <leon@is.currently.online>
Date:   Sun Sep 10 18:17:37 2023 -0400

     Add Hello World application
```

- Prof. Levy’s lecture will go into the details of this naming scheme

git 101

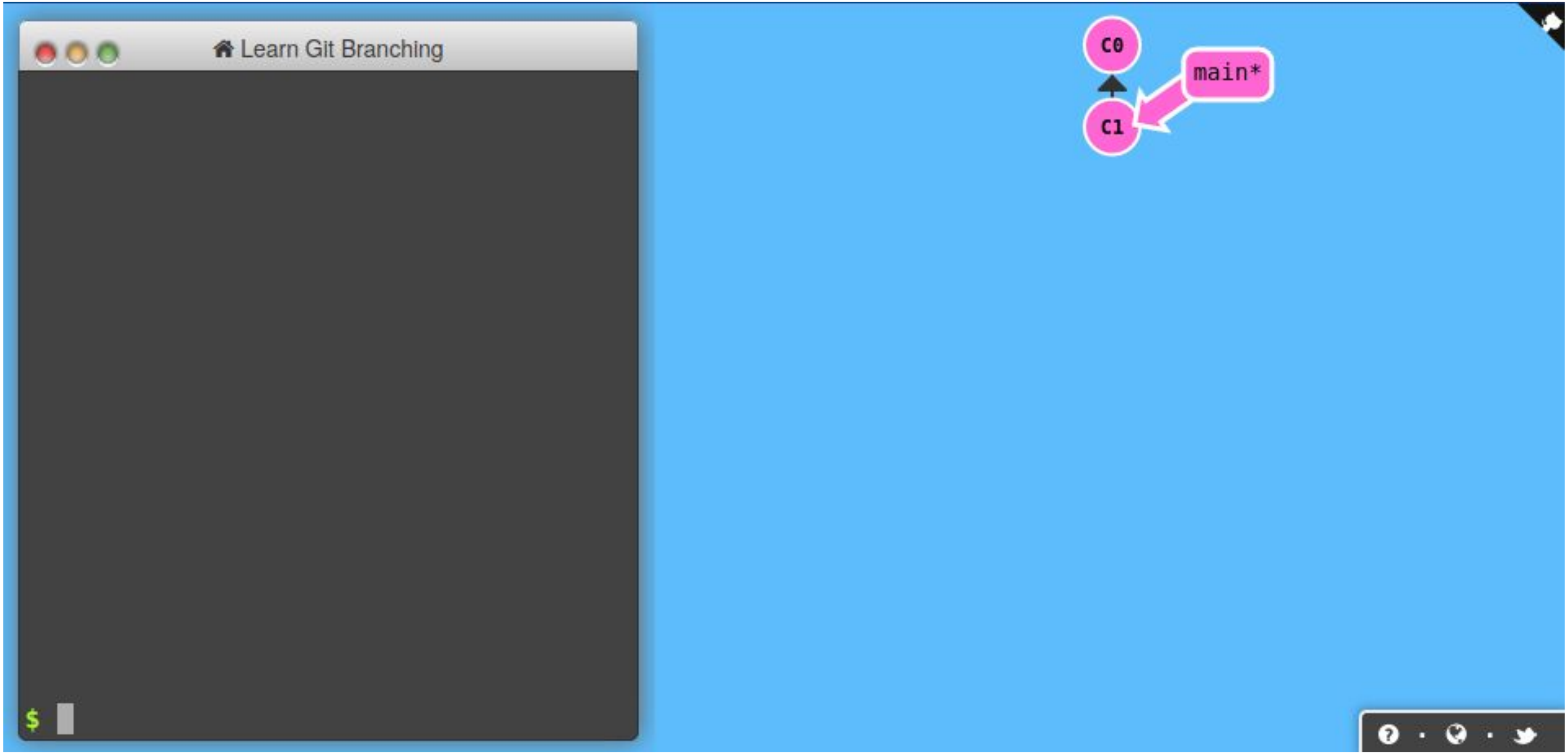
- Let's create a second commit!
- How does git know what *changed* in this commit?
    - git records the commit ids of the predecessor(s) of a commit

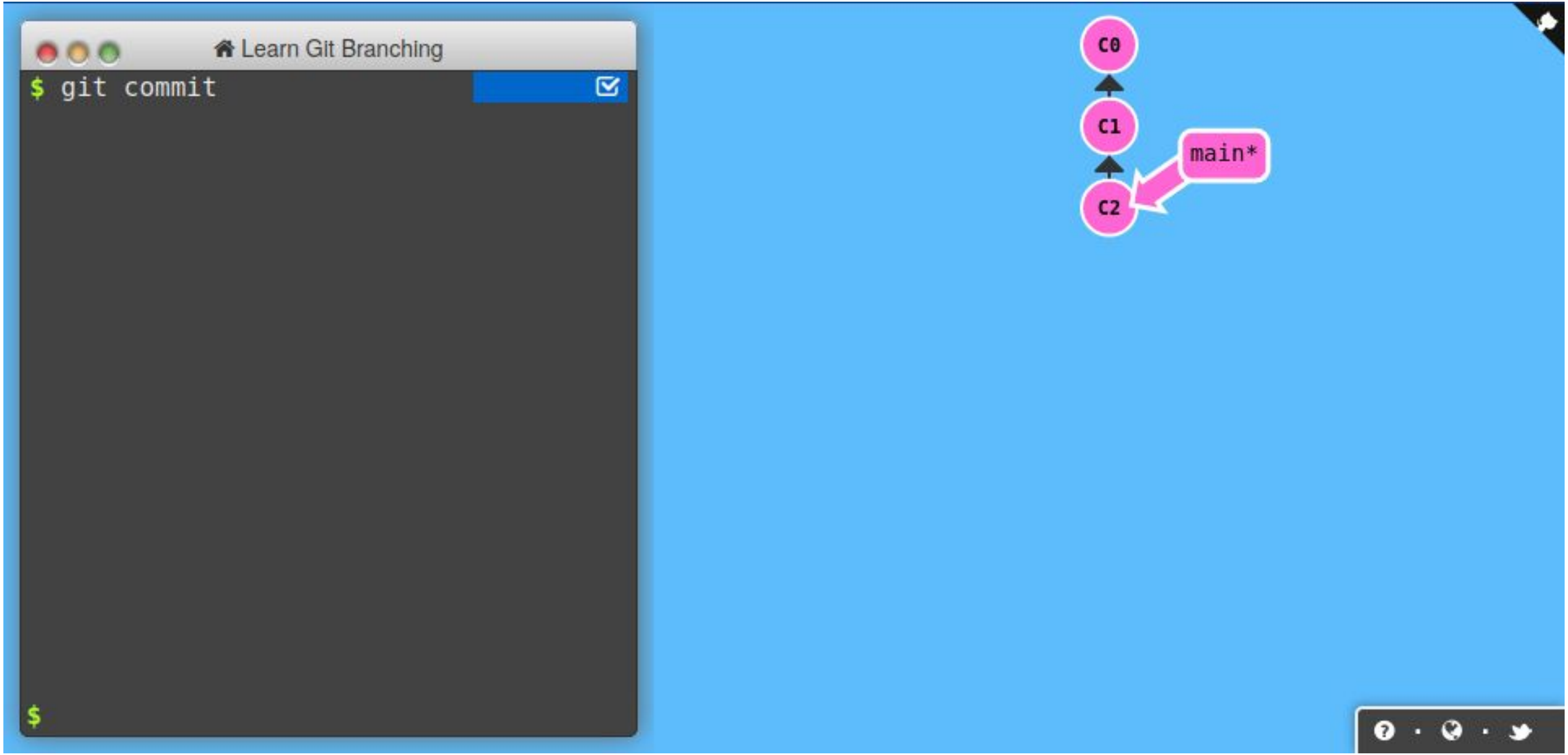
```
leons@silicon ~/c/cos316-repo (main)> git show --pretty=raw
commit 96758501677093f6ea8ce03f9debee0483e5f448
tree da837819f3f9b869299db74b932e88f136f667ae
parent fa937364b216ce78a07356b096618fbf85eca523
```

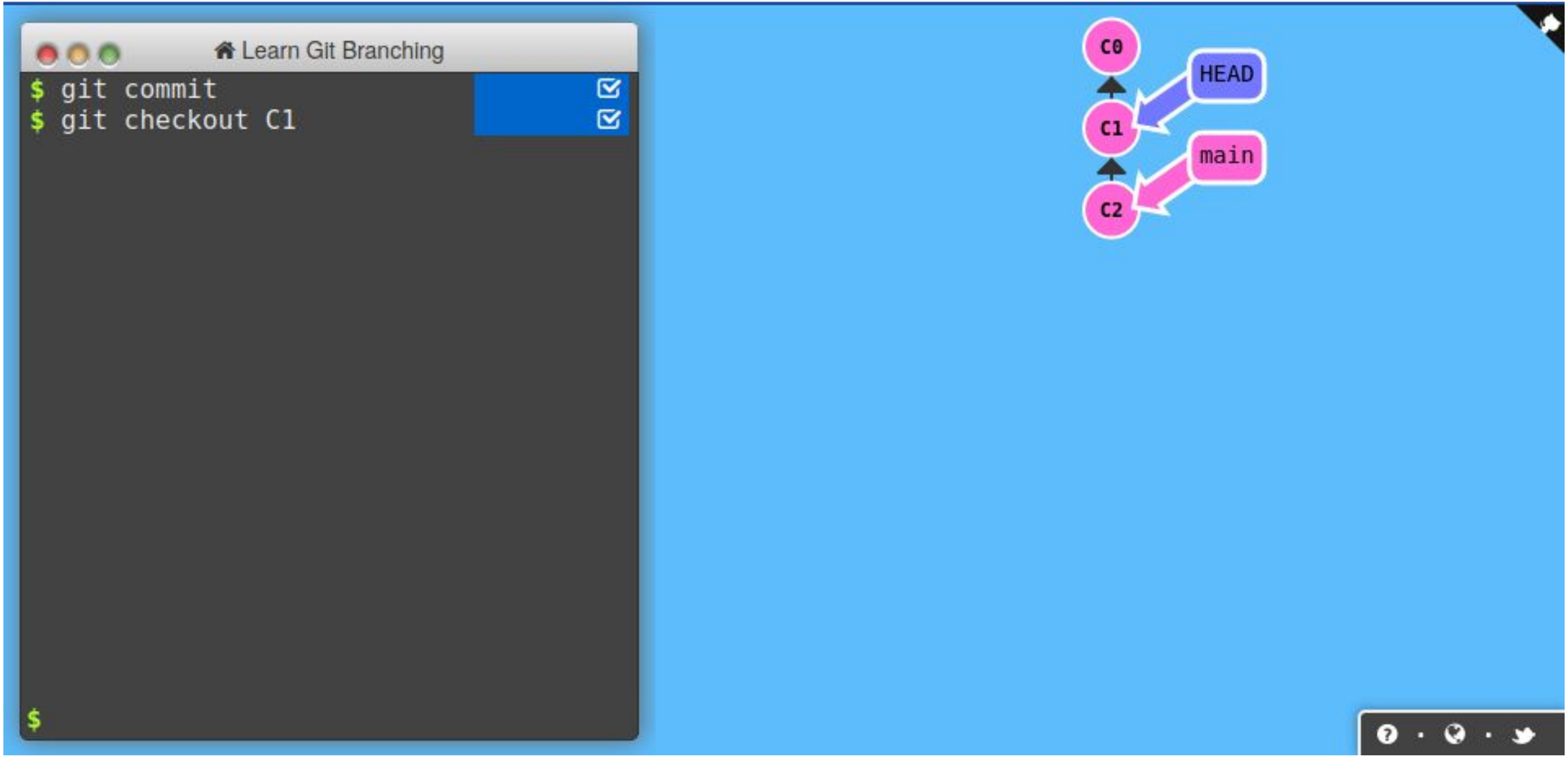
    - Creates a traversable version history, viewable with git log --graph

```
leons@silicon ~/c/cos316-repo (main)> git log --graph
* commit 96758501677093f6ea8ce03f9debee0483e5f448 (HEAD -> main)
| Author: Leon Schuermann <leon@is.currently.online>
| Date:   Sun Sep 10 18:49:02 2023 -0400
|
|     Make greeting more specific
|
* commit fa937364b216ce78a07356b096618fbf85eca523
  Author: Leon Schuermann <leon@is.currently.online>
```

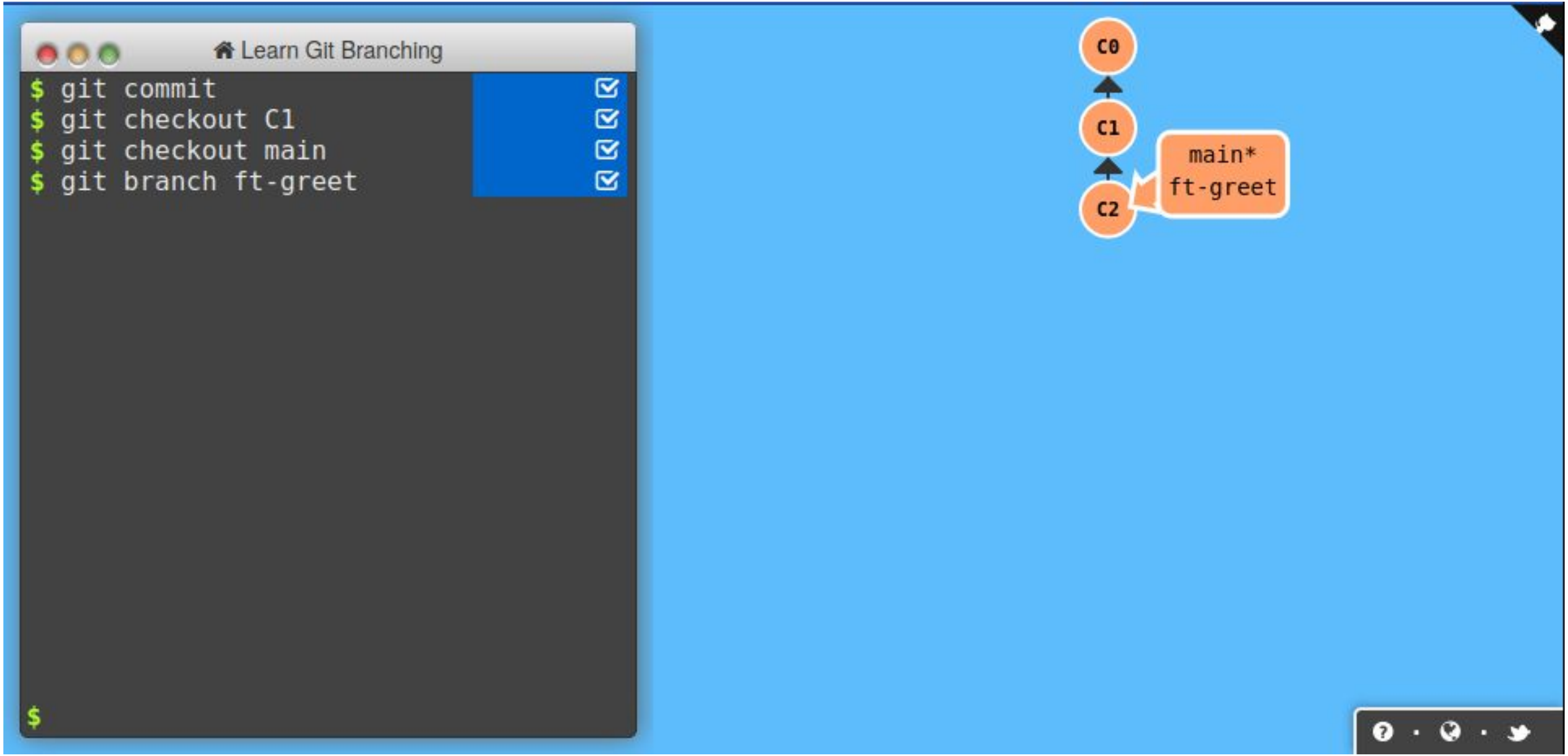
learngitbranching.js.org

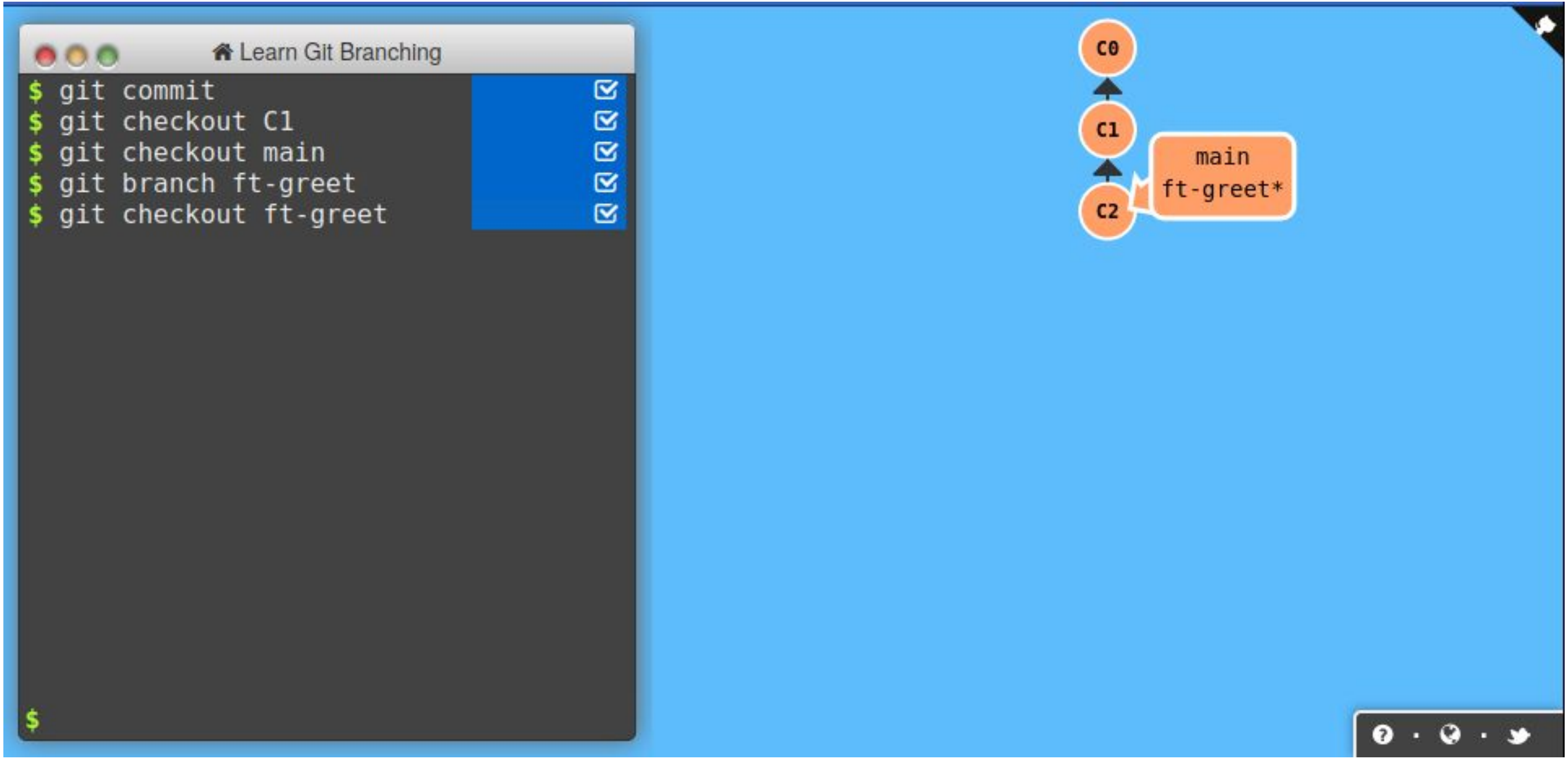

Learn Git Branching
$
C0
C1
main*

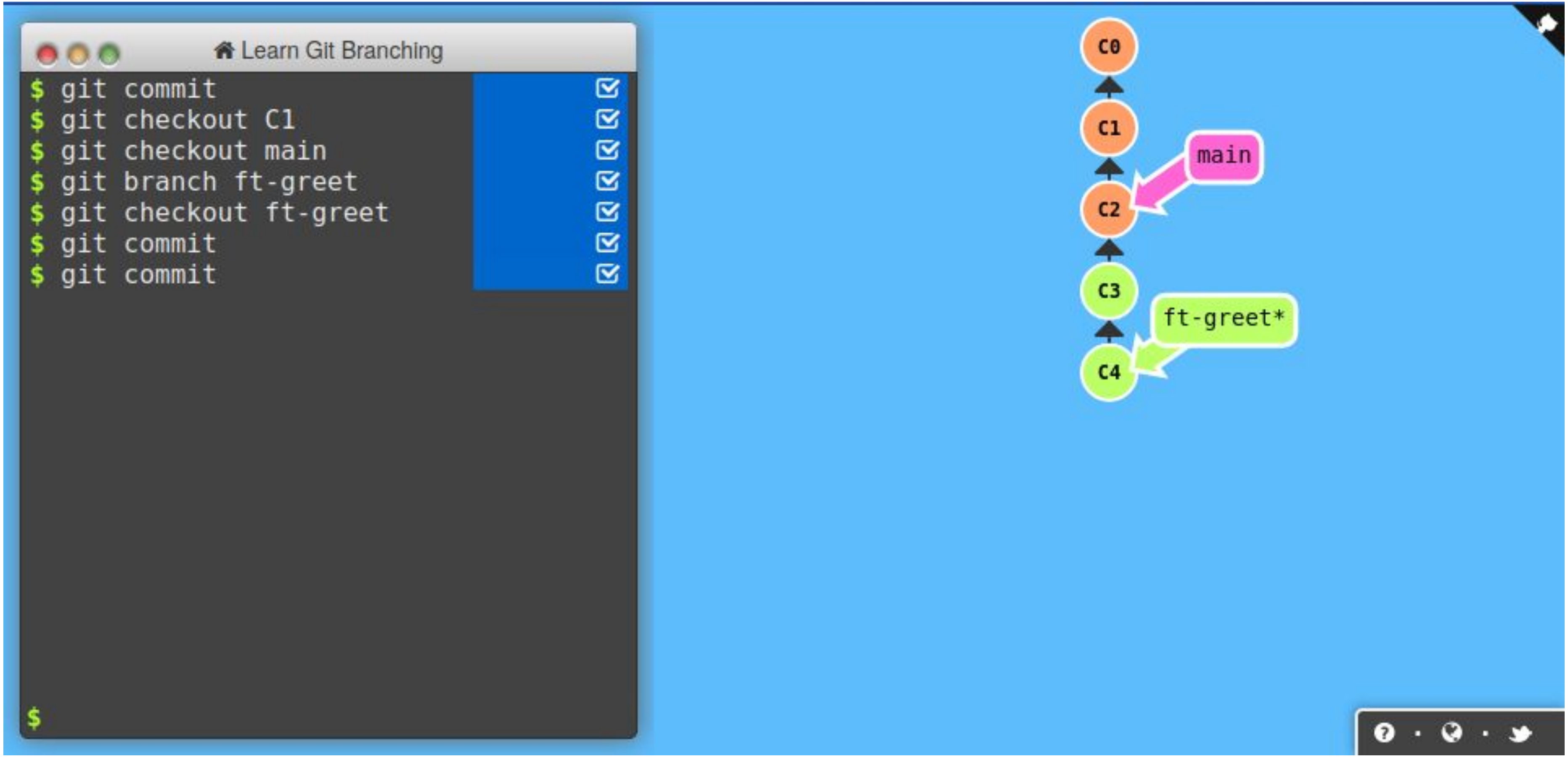

Learn Git Branching
$ git commit
$
C0
C1
C2
main*



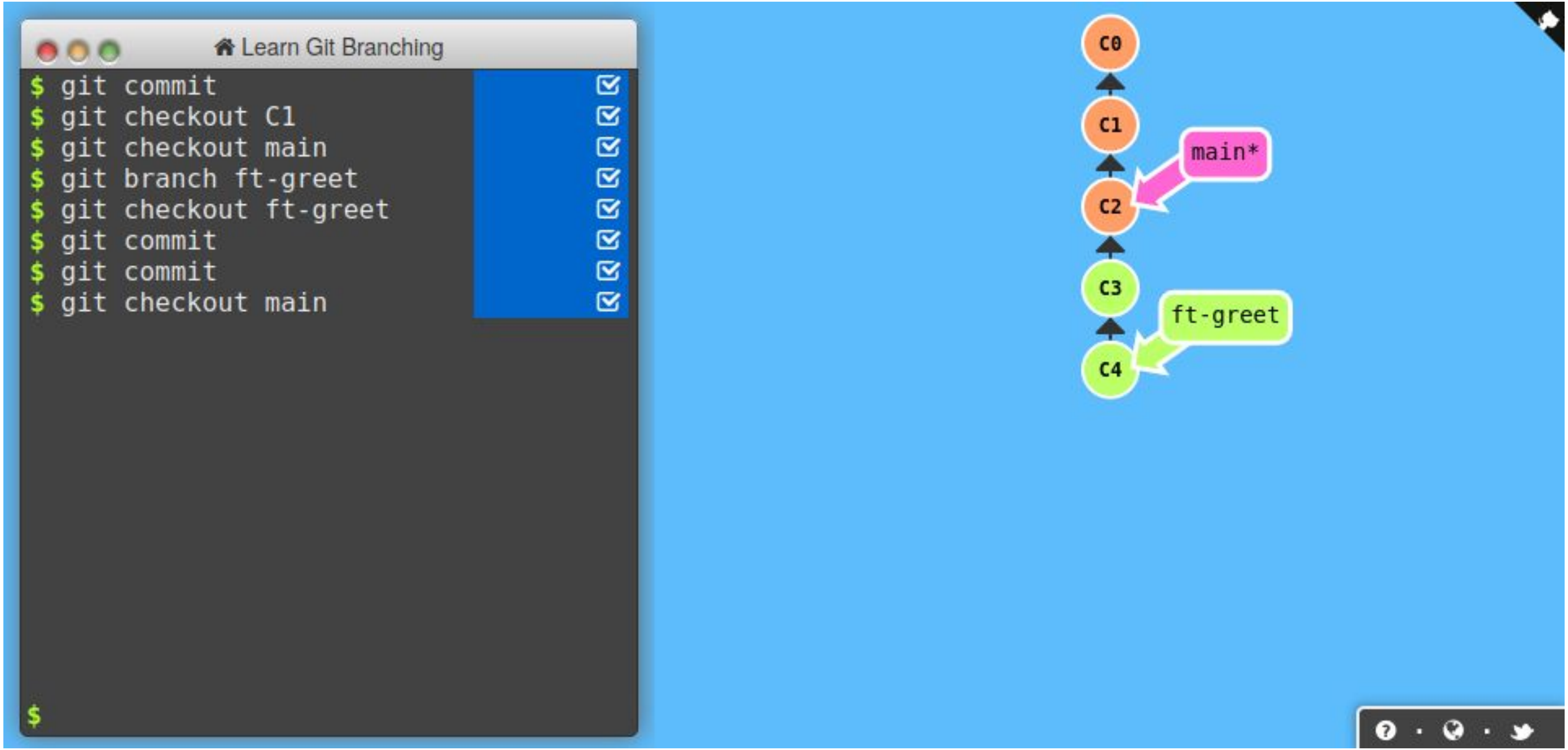
Learn Git Branching
$ git commit
$ git checkout C1
$
C0
C1
C2
HEAD
main

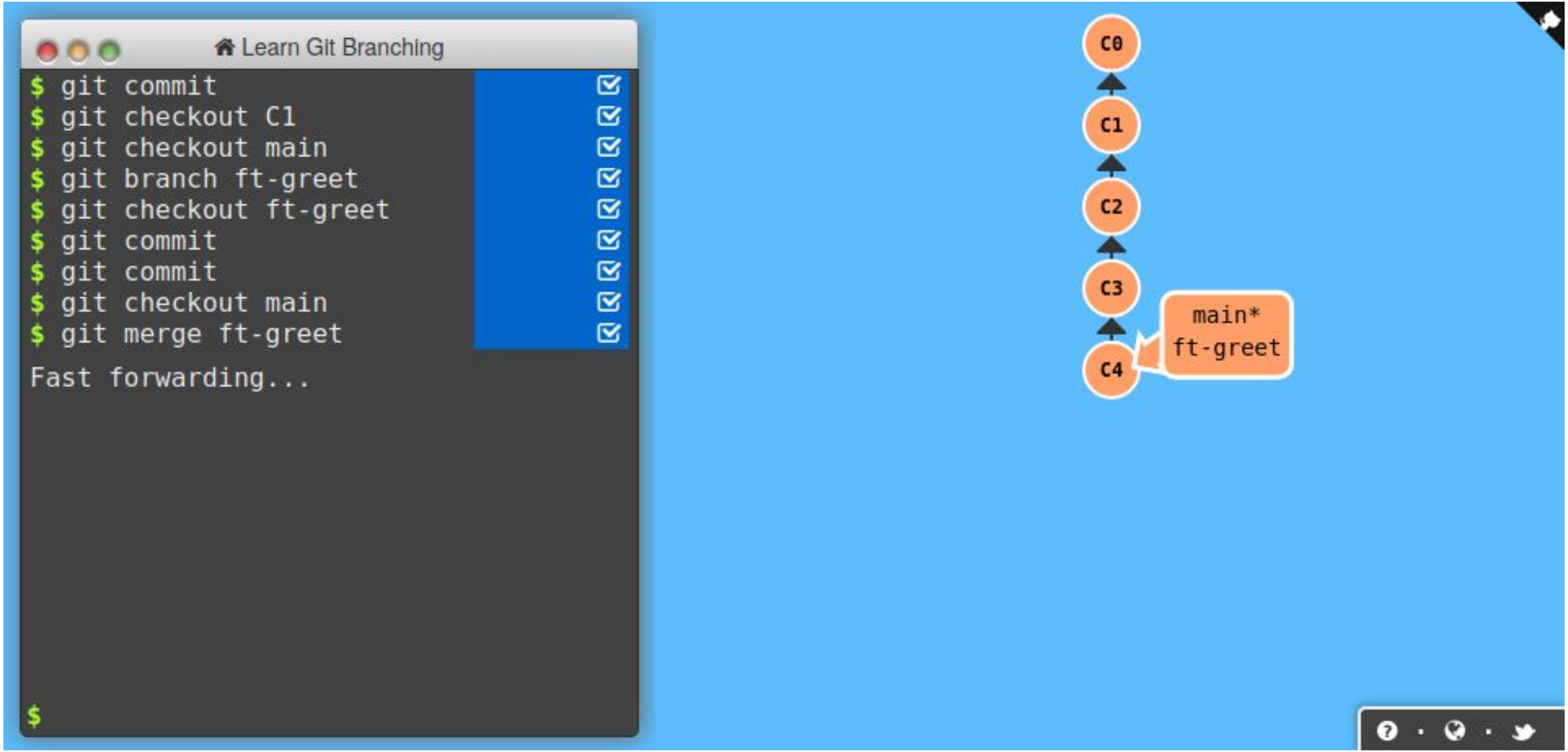

Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$
C0
C1
C2
main*
ft-greet

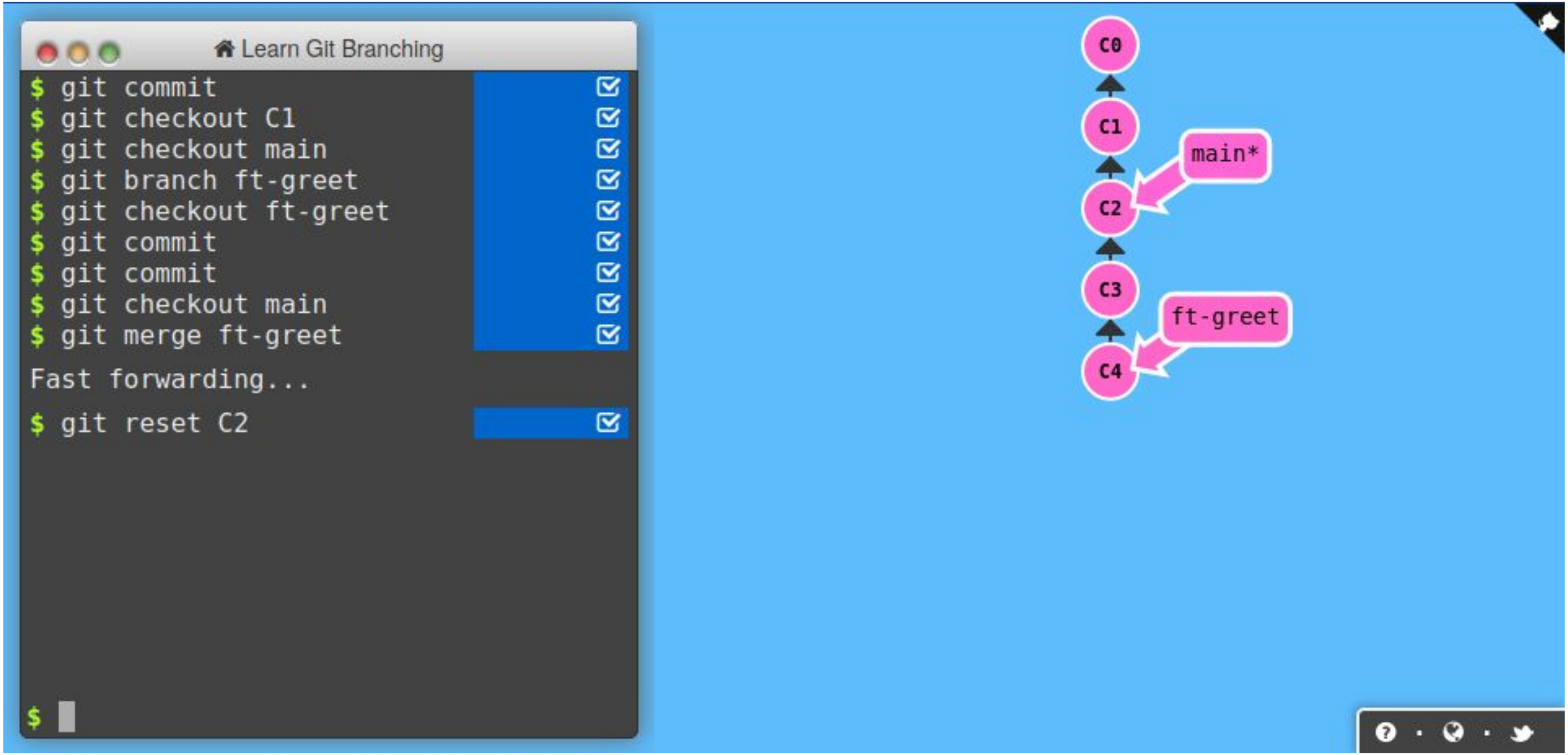

Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$
C0
C1
C2
main
ft-greet*

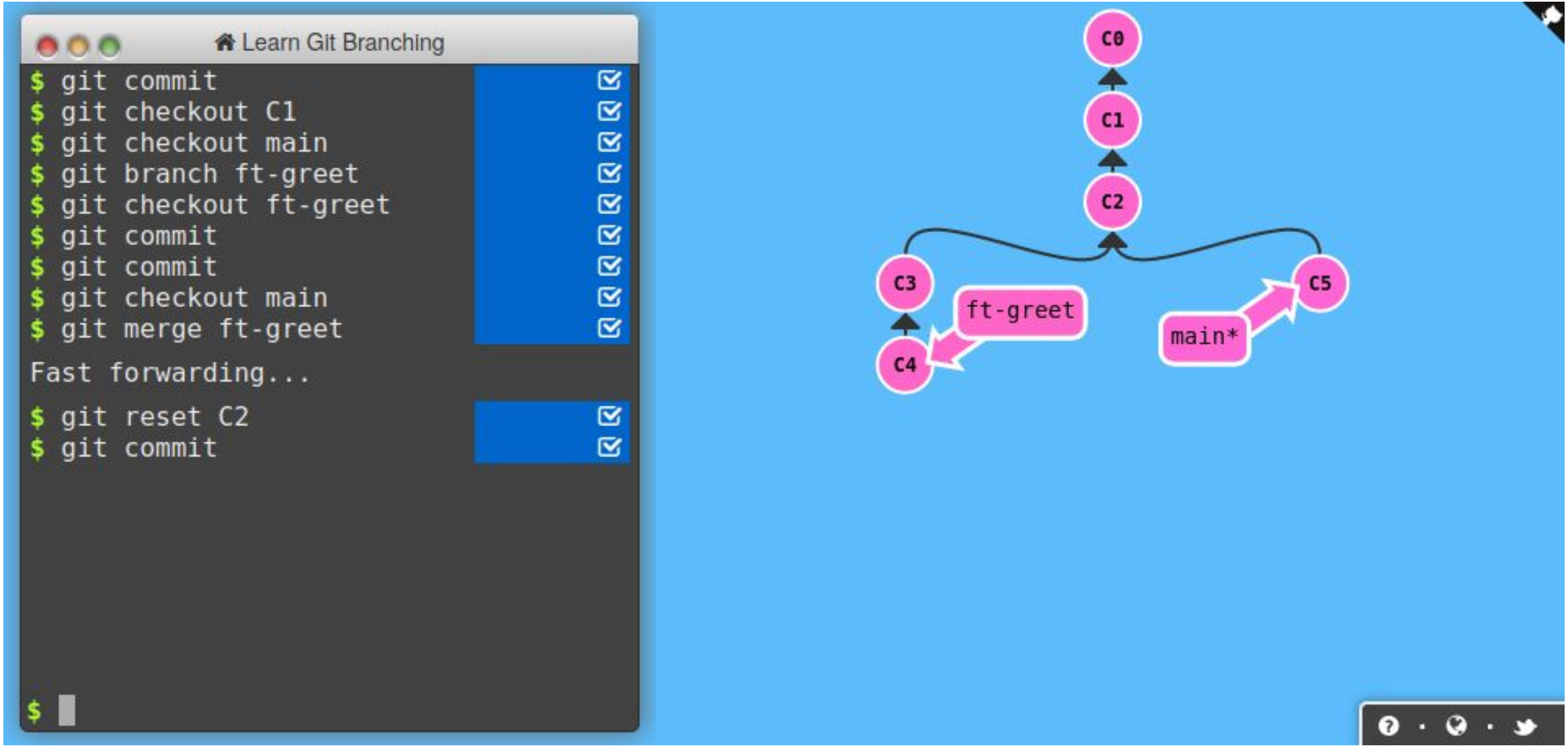


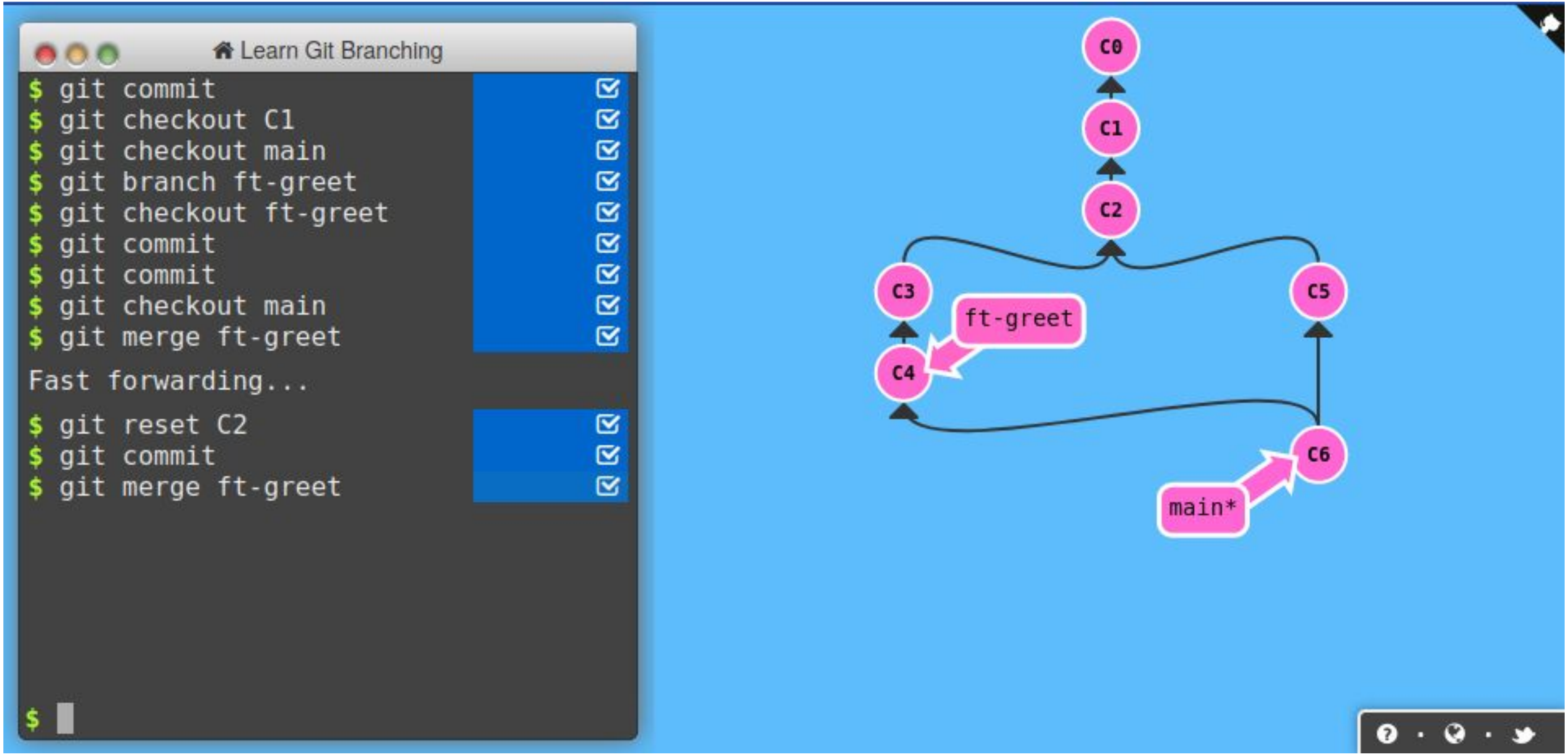
Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$ git commit
$ git commit
$
C0
C1
main
C2
C3
ft-greet*
C4


Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$ git commit
$ git commit
$ git checkout main
$
C0
C1
C2
C3
C4
main*
ft-greet


Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$ git commit
$ git commit
$ git checkout main
$ git merge ft-greet
Fast forwarding...
$
C0
C1
C2
C3
C4
main*
ft-greet

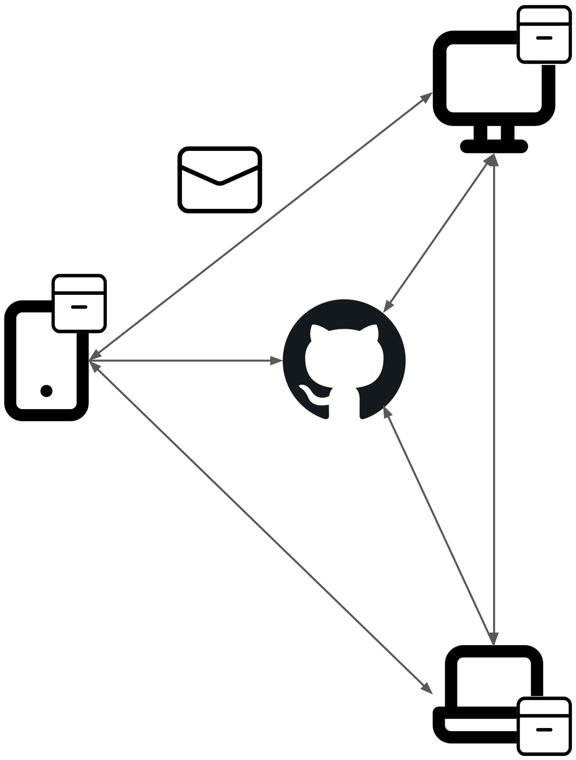

Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$ git commit
$ git commit
$ git checkout main
$ git merge ft-greet
Fast forwarding...
$ git reset C2
$
C0
C1
main*
C2
C3
ft-greet
C4


Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$ git commit
$ git commit
$ git checkout main
$ git merge ft-greet
Fast forwarding...
$ git reset C2
$ git commit
$
C0
C1
C2
C3
C4
C5
ft-greet
main*


Learn Git Branching
$ git commit
$ git checkout C1
$ git checkout main
$ git branch ft-greet
$ git checkout ft-greet
$ git commit
$ git commit
$ git checkout main
$ git merge ft-greet
Fast forwarding...
$ git reset C2
$ git commit
$ git merge ft-greet
$
C0
C1
C2
C3
ft-greet
C4
C5
C6
main*

Distributed Version Control with git

- git is a ***distributed*** version control system
- There can be many *clones* or copies of a repository
  - Sync commits, references, etc. between different copies
- Supports a fully decentralized workflow
  - No single “source of truth” server, as with e.g., Google docs
  - git works offline, without connection to a server or other clients
- *Software forges* (e.g., GitHub) provide git hosting
  - Just another (public) copy of your repository!
- Many different workflows for distributed git



Continuous Integration

- Automatic merges with git are great!
  - Enables working on features in parallel
  - Across multiple developers
- Merging codebases can break a system in subtle ways
  - E.g., you might rely on a function changed in a merged branch
  - Just because git does not detect a conflict, does not mean your program still works!
- This creates friction in the development process

Continuous Integration

- Continuous Integration (CI) is a development practice,
  → integrating contributions often (multiple times a day),
  → while building and testing automatically, on each merge.

  git takes care of the "automatic integration" part!

- One such system: GitHub Actions
  - Allows running arbitrary commands in "the cloud" (in a VM)
- This course's autograder is a "CI system"
  - Tests your code against a predefined set of test cases
    (You can't see the test cases though ☺)