COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Lecture 5
Tools of the Trade

Prof. Brendan Kochunas

9/18/2019

NERS 590-004

Outline

- Dynamic and Static Linking
- (?) Homework 1 Hands on
- Version Control
  - git
  - follow along demo
- Configure Tools & Infrastructure
  - CMake/CTest/CDash/CPack
  - Make
- More Hands on/Demo
  - Make and CMake of LAPACK
  - Example of CDash

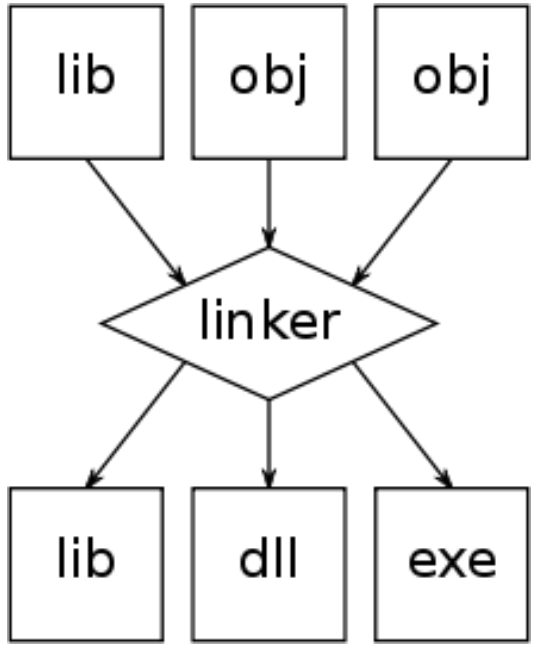
Learning Objectives

- Understand Static vs. Dynamic Linking
- Have an intuitive “mental” picture of how common git commands work
- Configure, Compile, and Link LAPACK

Static vs Dynamic Linking

What is Linking?

- Linking is the process of combining the various objects and libraries output from compilation into a single executable (or library or object).
  - May also include binaries (e.g. libraries) already installed on the system
- Sometimes performed by external program called by compiler (e.g. `ld`)
- Sometimes part of compiler (depends on the vendor)
- Key steps in linking are
  - *Resolving external symbols* that the linker uses to figure out how to piece together the executable
  - *Relocating load addresses* of various program parts (e.g. function addresses and variable addresses) to reflect the assigned addresses in the whole program.
- Linking can produce targets thatare ***statically*** linked or ***dynamically*** linked



Dynamic Linking vs. Static Linking

Static Linking

- Probably what you think of when you think "linking"
- Copy all binary code from all libraries and objects then package into a single executable image
  - Usually results in larger executable file sizes
- A little more portable since all the binary code is packaged together
- Requires all libraries that are linked to be static libraries (e.g. `lib<name>.a`)
- Sometimes a requirement on large clusters
  - Compute nodes and login nodes are different

Dynamic Linking

- Symbol resolution is delayed until executable is run
  - Executable code has undefined symbols
  - Requires all libraries that are linked to be dynamic libraries (e.g. `lib<name>.so`)
- Some advantages
  - For system libraries used by every program, no need to copy into every executable (e.g. `libc`)
  - If there is a bug in a library, and a new version of the library that fixes the bug is installed, all programs benefit.
    - Statically linked executables need to be re-linked
- Some disadvantages
  - Libraries that are updated that break backwards compatibility, might break your executable.
  - Need to have the correct environment.
  - Not necessarily portable, OS and environment need to consistent.

What link errors look like

Static Link Error

```
PROGRAM hello_main

WRITE(*,*) "Hello World!"
CALL some_undefined_routine()

ENDPROGRAM
```

```
$ gfortran -c hello.F90
$ gfortran hello.o -o hello.exe
hello.o: In function `MAIN__':
hello.F90:(.text+0x71): undefined reference to
`some_undefined_routine_'
collect2: ld returned 1 exit status
```

Dynamic Link Error

```
$ ./some_mpi_program.exe
./mpi_program.exe: error while loading shared
libraries: libmpi.so: cannot open shared object file:
No such file or directory
```

When you attempted to run the executable, The OS could not find the library using the information in your current environment

The command given to the linker did not include the library or object (or the correct path to the library or object) that defines the named symbol.

How to trouble shoot link errors

Static Link Error

- Most likely you are missing the correct entries on the following options passed to the linker:
  - `-l<library_name_with_symbol>`
  - `-L<path_to_library>`
- Could also be a typo in your source code
- Generally easy to resolve
  - If you know where the missing library is located.
- Can be difficult if you have no idea why the symbol is trying to be linked (where is it used, where is it defined
  - More likely to happen when you are linking third party libraries

Dynamic Link Error

- Most likely your environment is not the same as when you compiled
  - Check your environment
  - Environment variable is `LD_LIBRARY_PATH`
- Useful command: `ldd`
  - Shows you *exactly* what libraries are dynamically linked to your executable

```
$ ldd ./some_mpi_program.exe
        linux-vdso.so.1 =>  (0x00007ffcf2be8000)
        libmpi.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f4d12878000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f4d12c38000)
```

Dynamic ***Loading***: Linking in code at run time

- Start your executable *then load a library* into memory.
  - Use case is “plugins”. An example might be linking proprietary correlations for material properties.
  - Can be done interactively. User could specify library name and function name as an input.
  - Challenging to list “available symbols” in library, although this can be done. But basically need to know what routine you want to call
- In Linux requires “`dl`” library.

```
#include <dlfcn.h>

void* sdl_library = dlopen("libSDL.so", RTLD_LAZY);
if (sdl_library == NULL) {
   // report error ...
} else {
   void* initializer = dlsym(sdl_library,"SDL_Init"); //extract library contents
   if (initializer == NULL) {
      // report error ...
   } else {
      // cast initializer to its proper type and use
      typedef void (*sdl_init_function_type)(void);
      sdl_init_function_type init_func = (sdl_init_function_type) initializer;
   }
}
```

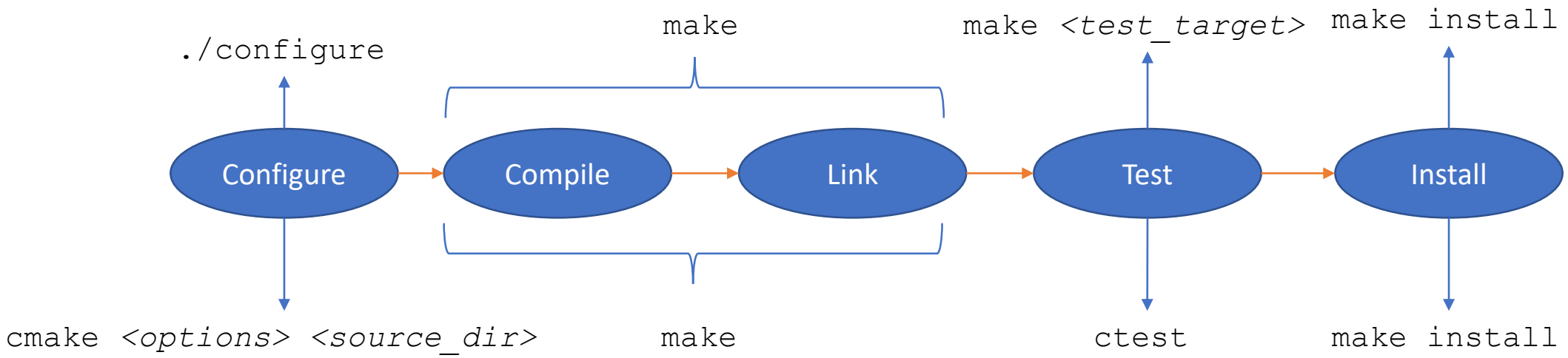
https://en.wikipedia.org/wiki/Dynamic_loading

Multi-language Programs

- **The key is linking!**
  - Linker does not care what high-level language produced your object code. It could have been generated from Fortran or C or C++.
    - Linker just has to resolve symbols in object code.
  - Well one subtlety, you must have a compatible application binary interface (ABI)
    - Usually not an issue unless you are compiling on one machine and linking on another.
- If a programming language or environment (e.g. Python) supports linking of C interfaces than you can link any code that provides a C interface
  - Most languages support C interfaces (because they were probably implemented in C or the compiler was)
  - Therefore, *C is the de-factor language of interoperability*.
- By “C interface” I mean a binary symbol that is producible from the C high-level language and a C compiler.

Summary: Using the Toolchain

Autotools



CMake

Version Control

What is Version Control?

- A way of tracking detailed changes in source code over time.
  - e.g. what changed? when did it change? who changed it?
- Version control is an ***essential*** component to software development.
  - Has been used by software developers for decades
- Version control is generally performed using an external program.
- Version control is applied to a “repository”.
  - Source code lives in one (or more) *respositories* (e.g. repos) available to team members/contributors.
- A repo is a computational scientist’s laboratory notebook.
  - Like a laboratory notebook, it is only useful if it is used properly.
- Also known as: “source code control”, “revision control”, “source code versioning”

Slide source: Best Practices for HPC Software Developers – Session 3: Distributed Version Control and Continuous Integration Testing

What version control does

- Establishes a comment context for code contributions and the exchange of ideas
- Establishes a chronological sequence of events
  - A single change is commonly referred to as a *commit*
- Serves as “truth” for a software project
- If you don’t have a “tangible” common reference for your source code, *there is nothing for your team to discuss*.
- Results from uncontrolled code are (generally) *not reproducible*.
- Recall your most frustrating document-sharing experience...
  - ...and imagine it continuing for months... or years with the people involved changing and the document getting larger and larger.
  - (it doesn’t work)

Utility of Version control

- What if I program by myself?
  - Still offers same advantages.
    - Try to remember the steps you took to complete a homework assignment in high school.
  - Working on different machines is no problem.
  - If you eventually want to collaborate with someone then there's no extra work!
- Can I use version control for other things?
  - YES!
  - Most version control programs are excellent It's great for text files. Working with binary files, its not so good.
  - You may not think its worth the extra effort now, but your future self will thank you.
    - Commonly used for LaTeX documents.

Version Control and Versioning

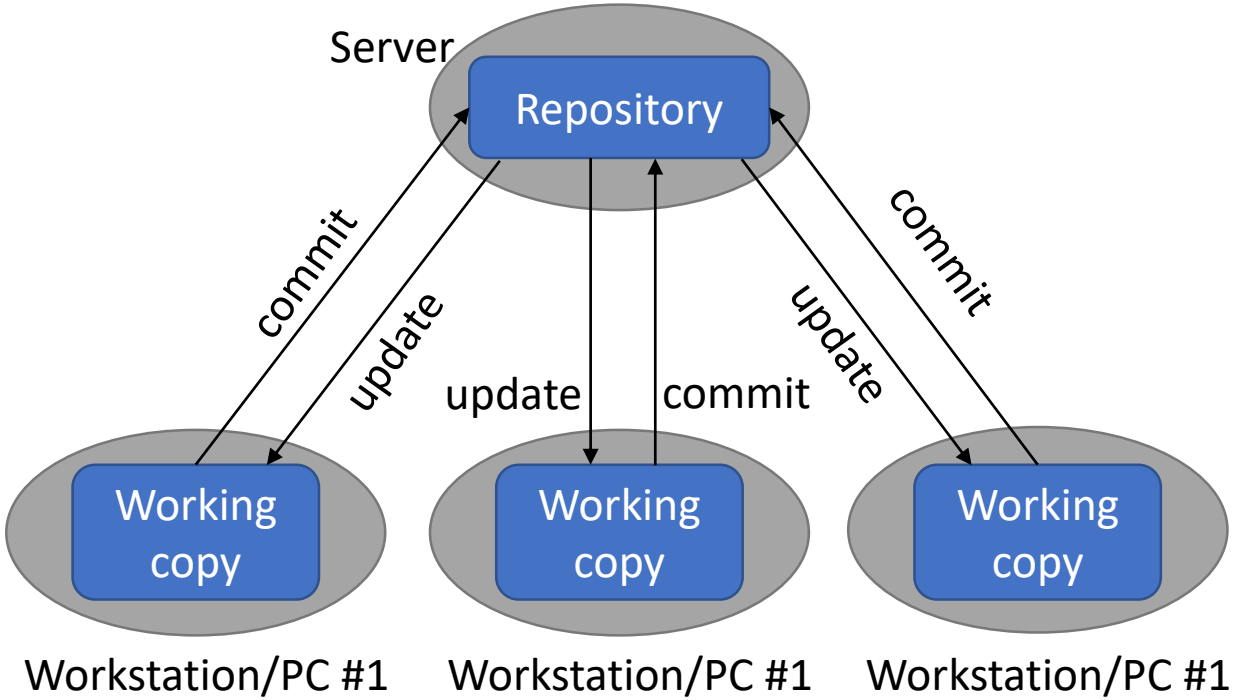
- Version control allows you to define and publish a specific version of the code. (e.g. one specific commit)
- A popular description for how to describe a version and what it means is defined here:
  - http://semver.org/
  - Short for “Semantic Versioning”

Short Definition from http://semver.org/

- Given a version number `MAJOR.MINOR.PATCH`, increment the:
  - `MAJOR` version when you make incompatible API changes,
  - `MINOR` version when you add functionality in a backwards-compatible manner, and
  - `PATCH` version when you make backwards-compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the `MAJOR.MINOR.PATCH` format.

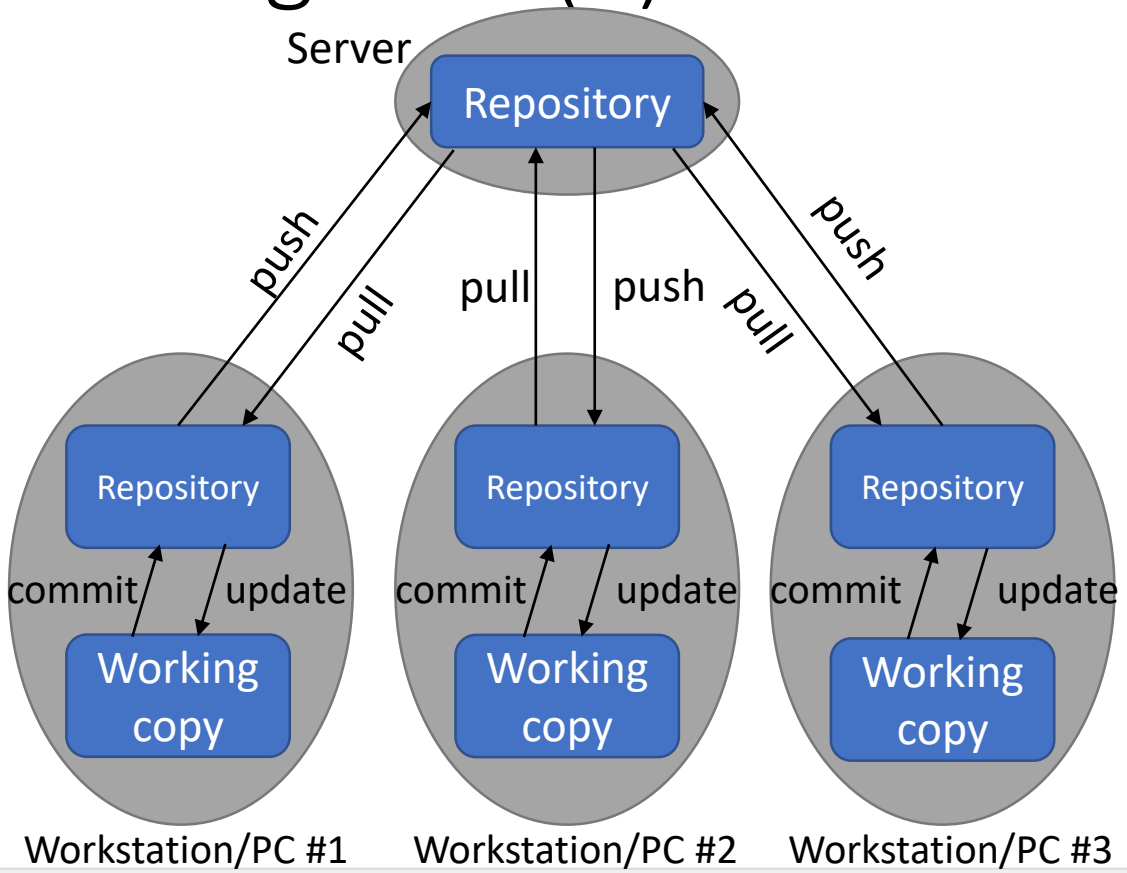
Models for Version Control Programs (1)

- Simple: “Centralized” Version Control
  - There is one repository containing the “master” version (or “trunk”) of the source code
  - Everyone syncs with this repository
    - e.g. *checks out* files, *changes* them, and *commits* these changes.
  - Requires that people must cooperate/coordinate to make sure their changes don’t conflict with each other.
  - Limited in capability to create development branches.



Models for Version Control Programs (2)

- Modern: "Distributed" Version Control
  - Everyone has an entire copy of the repository (and history!)
  - There is a "main" repo agreed upon by convention
  - People typically work in development branches with isolated changes until ready to merge
  - Allows for more flexibility for design and development procedure



Open Source Tools for Version Control

Centralized Version Control

- Concurrent Version System (CVS)
  - One of the first. Don't recommend you start here.
  - Should not need to work with it
    - There are many tools to migrate a CVS repository to a newer one.
- Subversion (SVN)
  - Modern successor to CVS
  - Supports branching

Distributed Version Control

- git
  - Probably the most popular version control program
    - Written by same person who wrote linux.
  - Several programs built around git offering different interfaces
    - gitlab, github, gitorious
- Mercurial (hg)
  - favors ease of use
  - syntax similar to subversion

Version Control Disclaimer

- It's a tool, and its only as good as its user.
- It does not define a *development process*
- Up to you to choose an approach that best works for you and your team
- Identifying and using good software development processes is hard.
  - That's why we'll talk about it in a future lecture

http://xkcd.com/1296/

A little about git

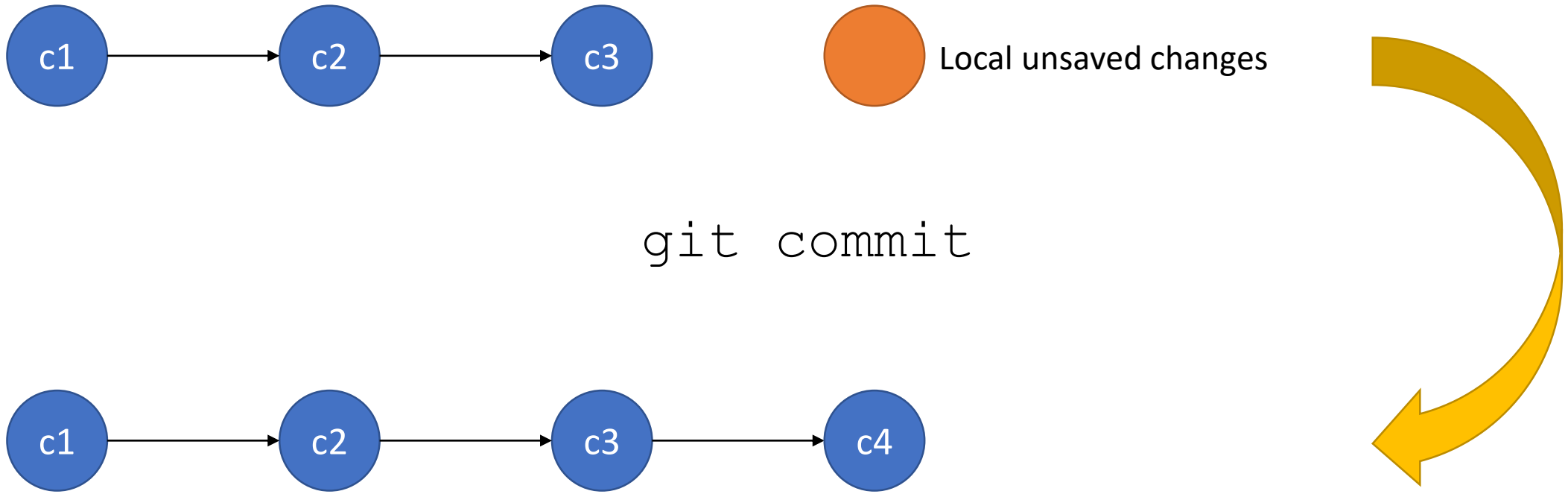
- Initially released in 2005
  - original author was Linus Torvalds (the linux guy)
- Difficult to learn without putting in some time
  - That's what the demo is for!
- It is difficult to learn without understanding the underlying concepts.
- Teams that use git well often have an expert.

https://xkcd.com/1597/

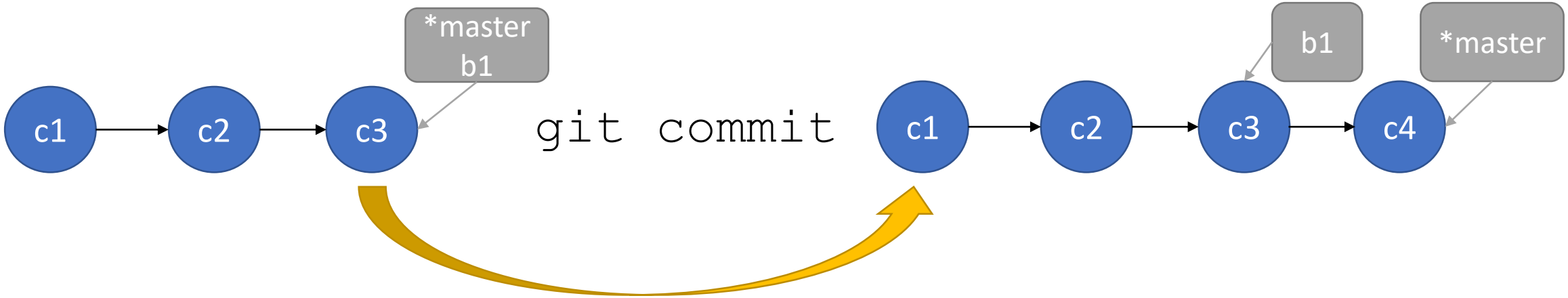
Common Nomenclature in Git

- *Repos* – repositories. the full history of the project
- *Clones/cloning* of repos – making a copy of
- *Commits/committing* within repos – making code changes
- *Branches/branching* within repos – isolated development
- *Remotes* – references to other repos on other machines
- *Pulls/pulling* – incorporating changes from another repo to your local repo
- *Pushes/pushing* – publishing changes from your local repo to another repo
- *Revisions* – also known as commits, can be referred to by a the SHA-1 (e.g. 1e95a651f4aeea1aa00173347f254dfb93ae350a)
- *Workspace* – local state
- *History* – the graph, specifically a directed-acyclic-graph (DAG)

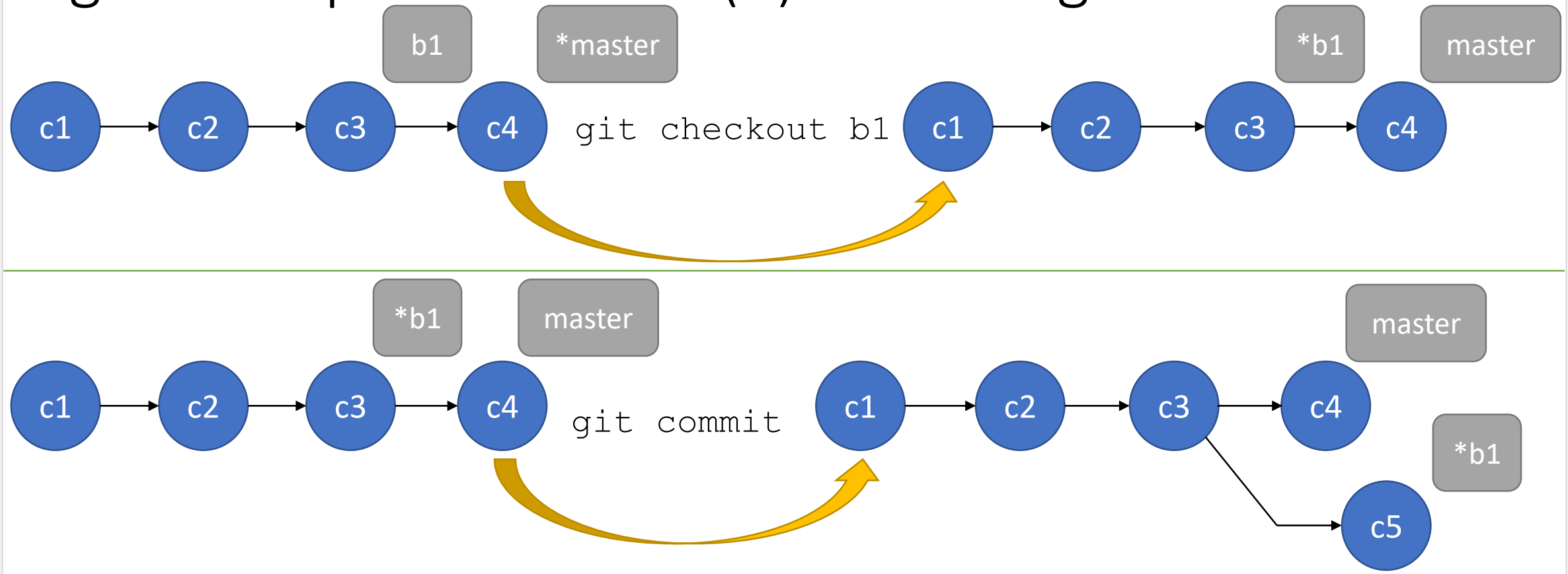
git concepts: commits



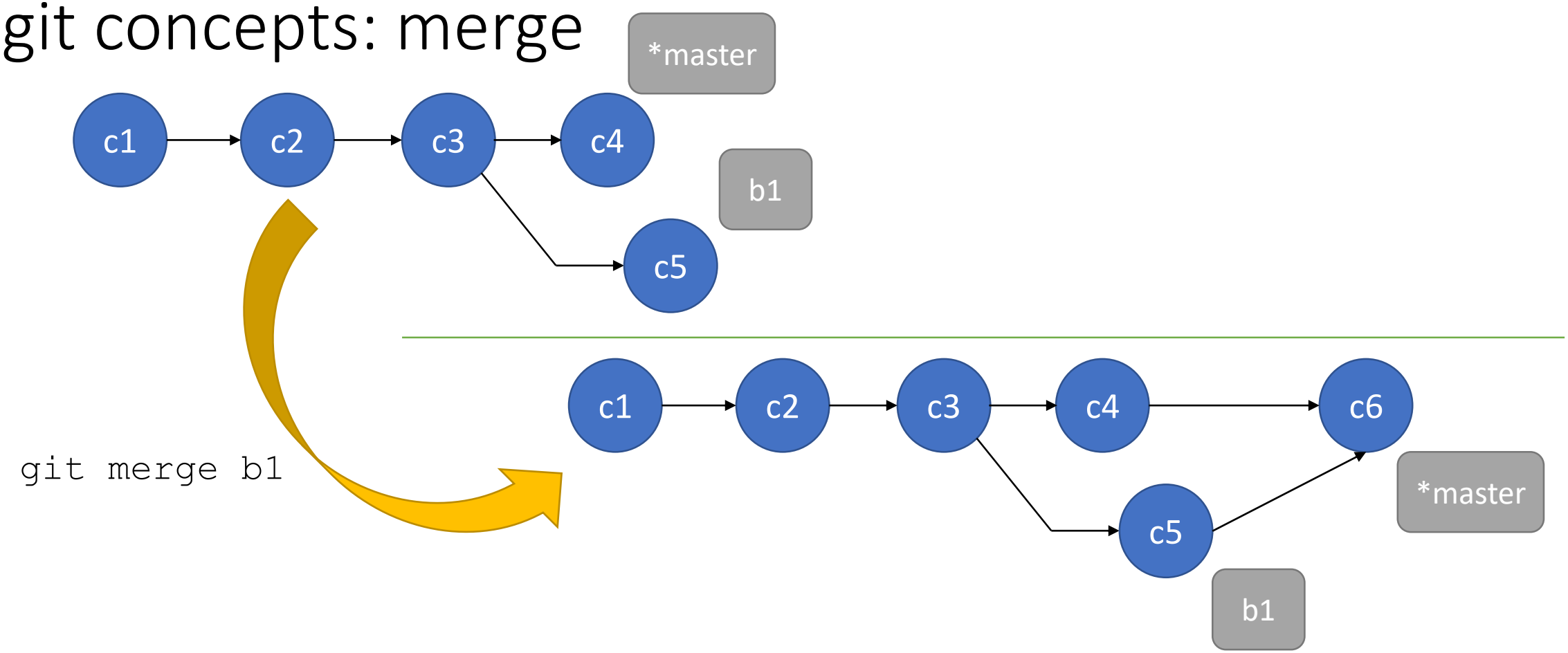
git concepts: branches (1) – creating a branch



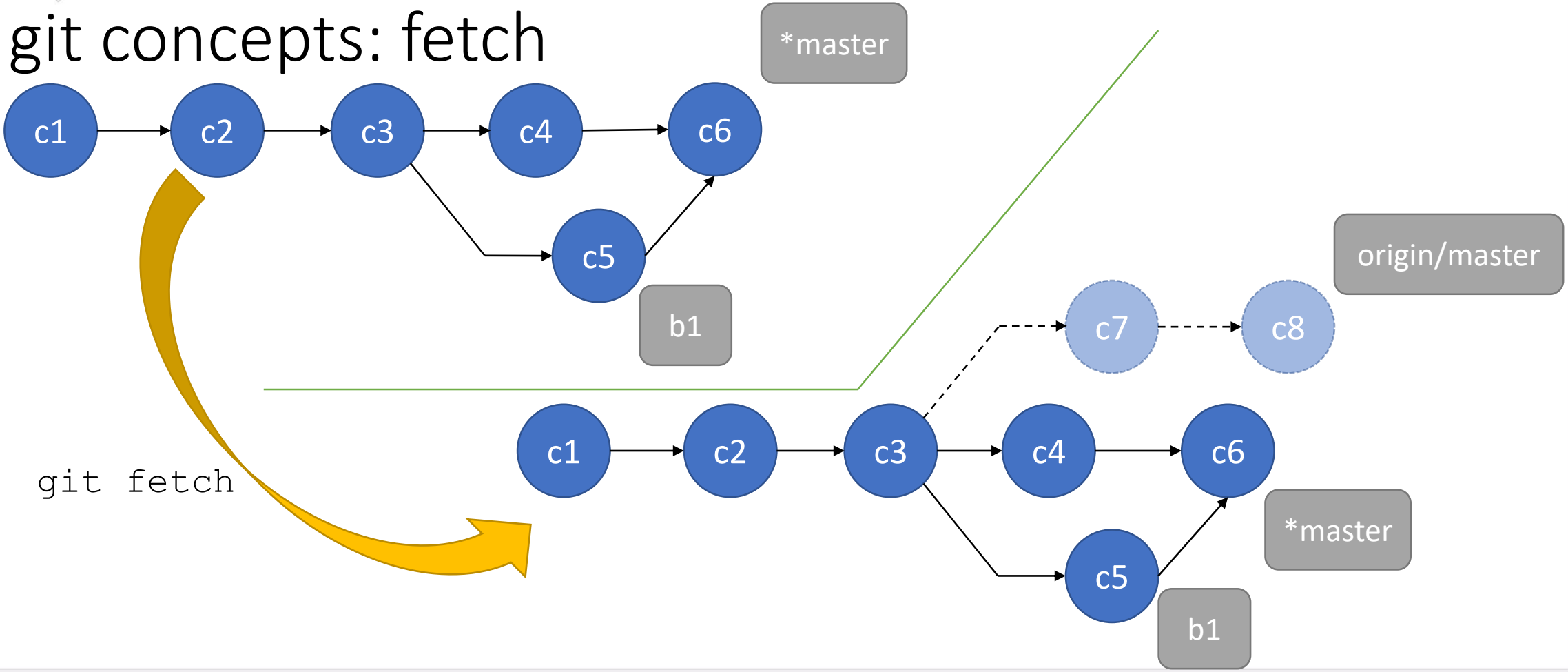
git concepts: branches (2) – working on a branch



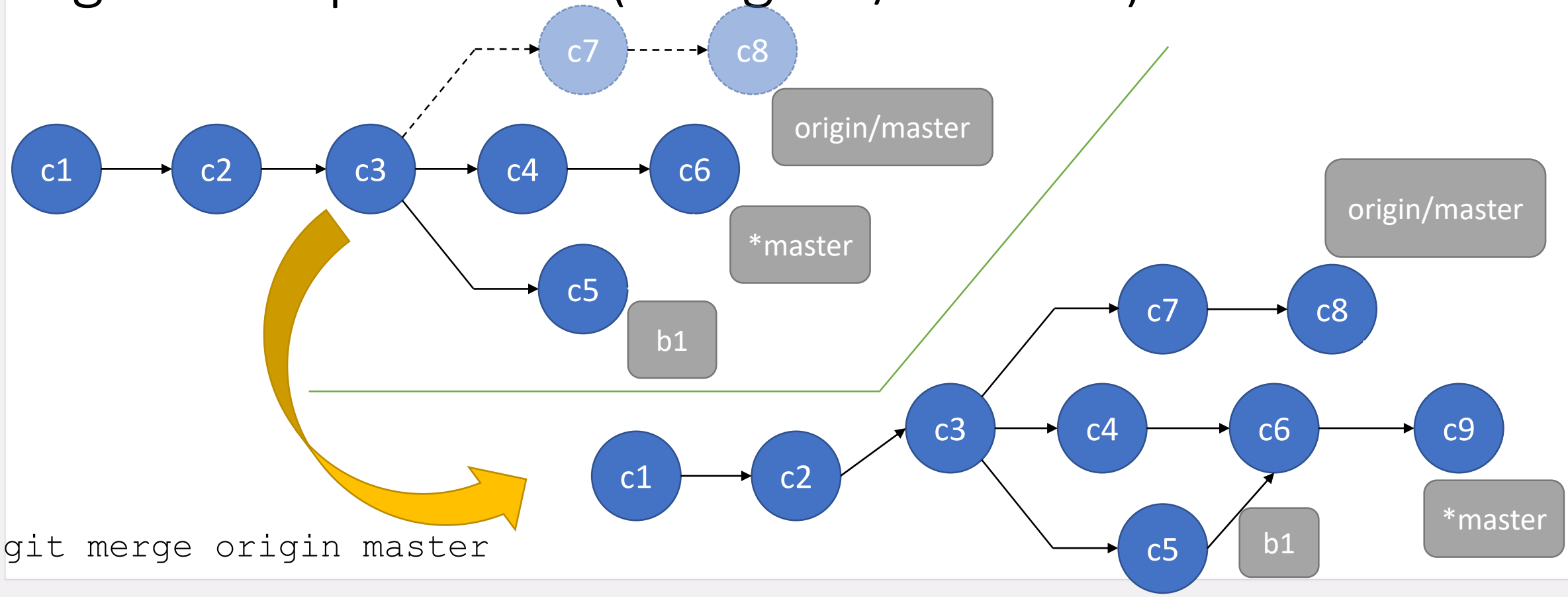
git concepts: merge



git concepts: fetch

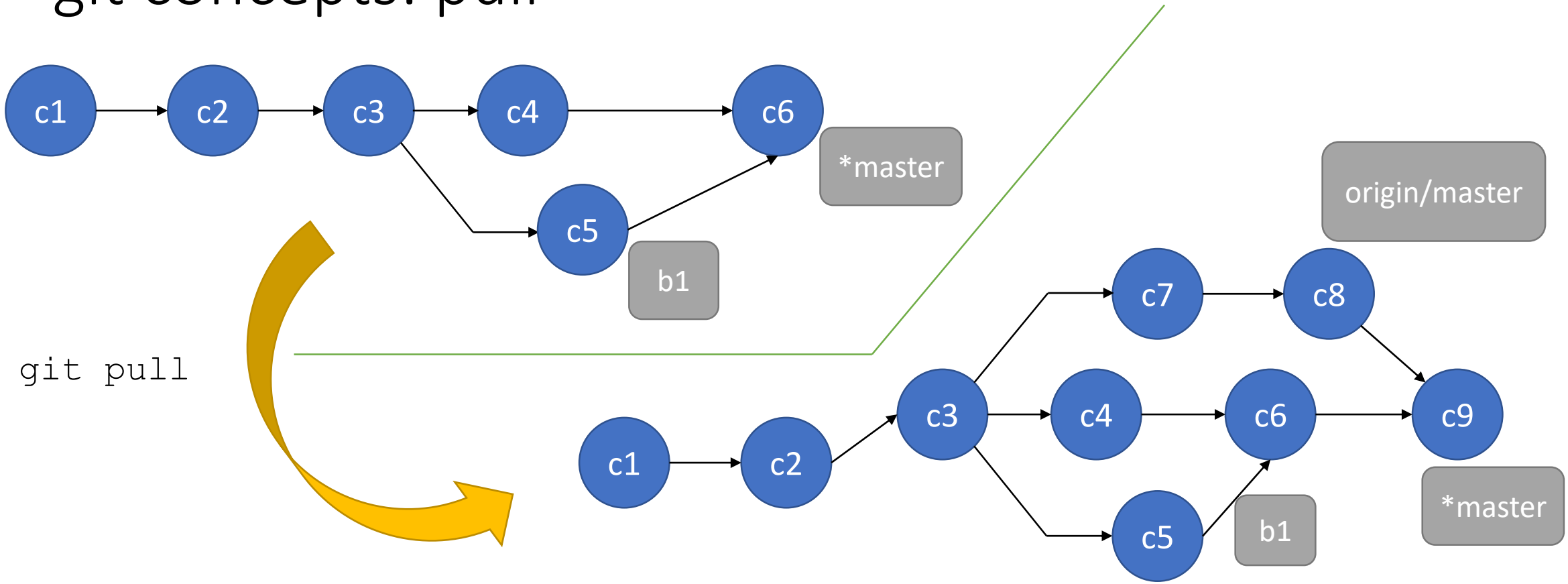


git concepts: fetch (merge w/ remote)



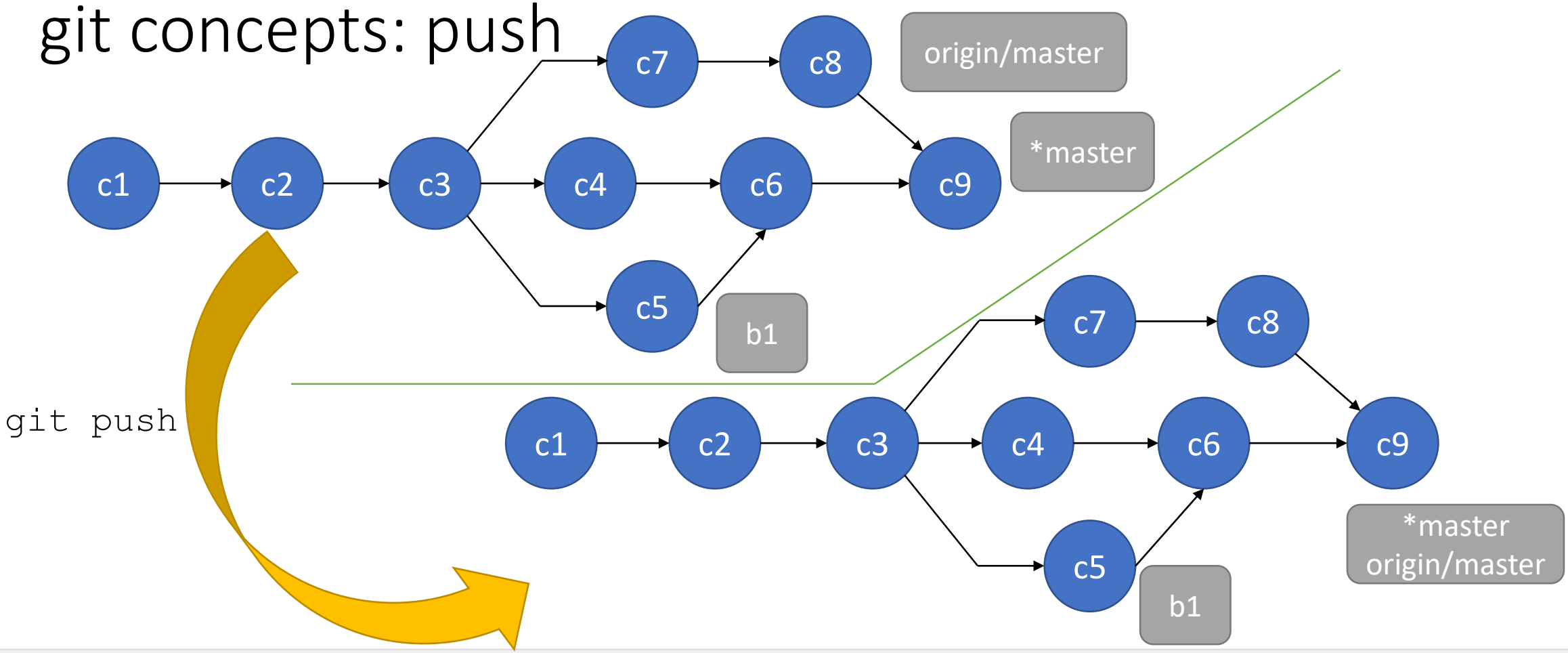

```
git merge origin master
```

git concepts: pull



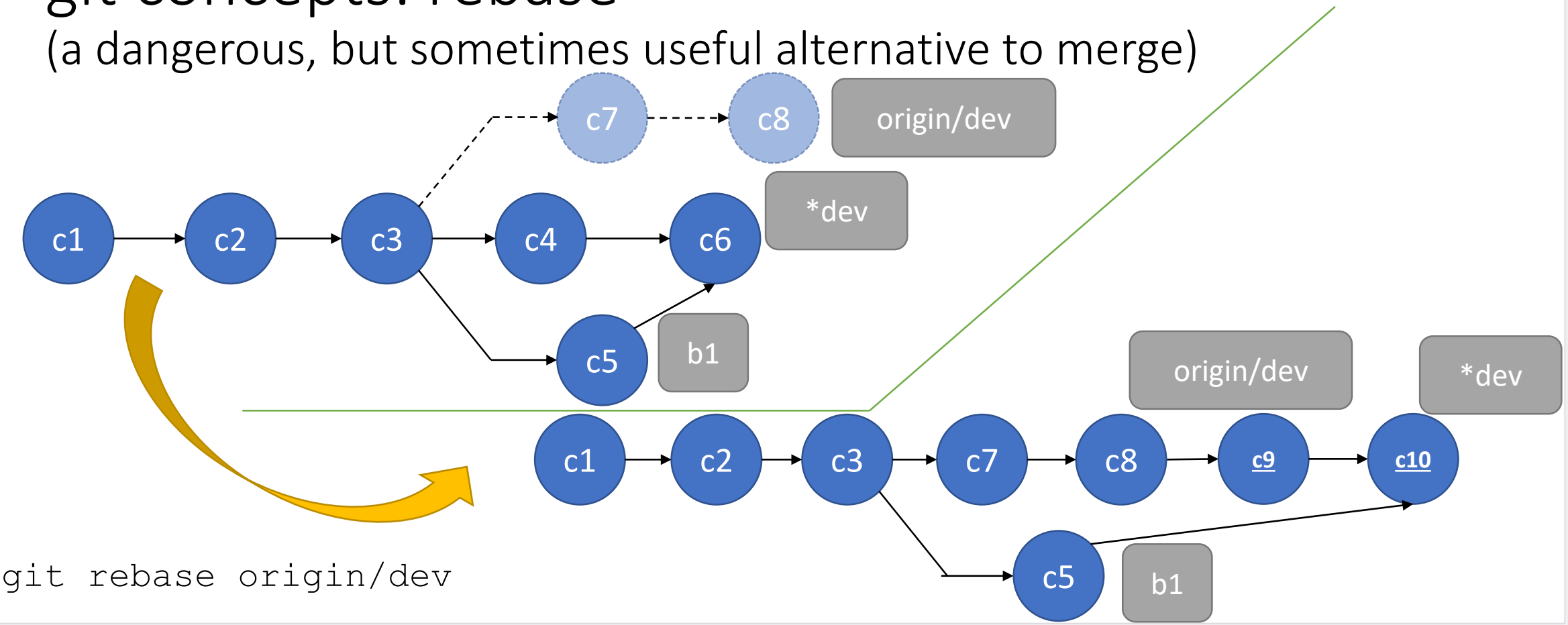
`git pull = git fetch && git merge`

git concepts: push



git concepts: rebase

(a dangerous, but sometimes useful alternative to merge)




```
git rebase origin/dev
```

rebase ***rewrites the history***.

Do not mix merge and rebase (unless you know what you are doing)

A little git hands on

http://learngitbranching.js.org/
https://ideas-productivity.org/resources/howtos/git-tutorial-and-reference-collection/

Infrastructure Tools

Now you know how to track changes
What's next?

What is it?

- Open source
- Cross platform (truly)
  - Supports multiple build tools, toolchains, & environments.
- Includes
  - CMake – configuration tool
  - CTest – testing tool
    - automates testing and collection and publishing of test results
  - CPack – generates installers
  - CDash – Web interface for viewing test results

Software Projects Using CMake

- CGAL
- Geant4
- GROMACS
- Trilinos
- VTK and Paraview
- zlib
- LAPACK
- HDF5
- Netflix

(GNU) Make

- Likely the most popular tool for defining executables and compiling software in Linux
  - Alternatives: Ninja, Tup, Gulp
- Standard in Linux/Unix
- CMake generates makefiles for you
  - It can also generate inputs for other build tools
  - Gives you fancy features
    - parallel, percentage complete, default targets, help, coloring.

- Makefiles must define targets using rules
  - Can also define variables, use include

```
target ... : prerequisites ...
          recipe
          ...
```

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
       cc -o edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o

main.o : main.c defs.h
       cc -c main.c
...

clean :
       rm edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
```

Examples from GNU Make manual: https://www.gnu.org/software/make/manual/make.html

CMake Example/Demo