CSE462/562: Database Systems (Spring 22)

Lecture 2: Introduction to Taco-DB & C++ primer

2/3/2022

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Taco-DB

- Taco-DB is a teaching-oriented DBMS developed for course projects
  - Also used at PSU this semester
  - Language: C++11
  - Build system: cmake
  - Architecture: x86_64 (no ARM, e.g., Apple M1, Microsoft SQ1/SQ2, etc.)
  - OS: Linux
    - it's known to work on Ubuntu 20.04, Fedora 35
    - and CSE student servers (CentOS) with a few package customizations
    - WSL2 or virtual machine with Linux are also ok

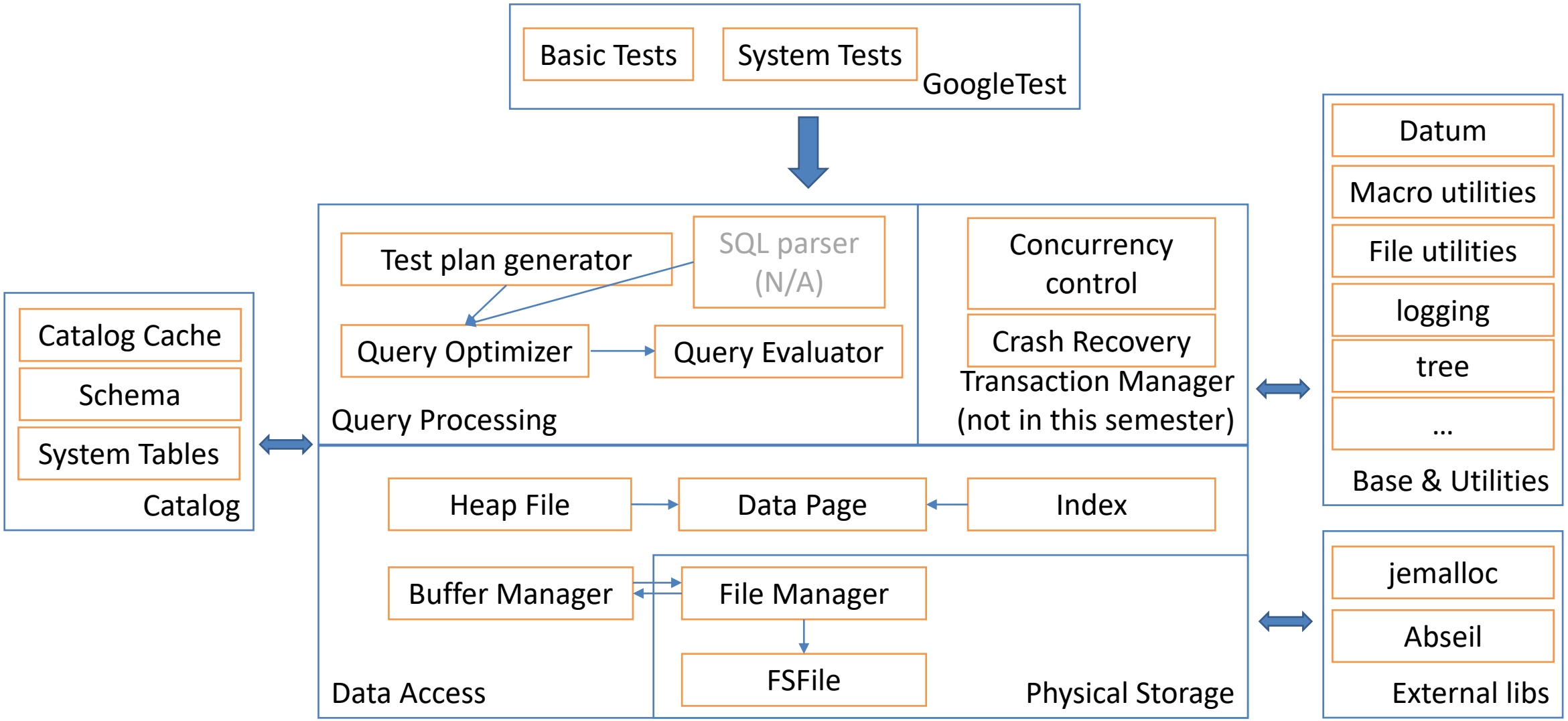
- Developed in house -- you won't be able to find solutions or references online
  - We expect you to keep it private indefinitely!
    - Please do not make your repository public or share with current and/or future students
    - It's ok if you disclose it to non-students (e.g., in job/PhD interview)
  - Send bug report to Piazza if any.
  - That helps us to improve and reuse it for future CSE462/562.

Prerequisites for the course project

- It's best if you know C/C++ and have some experience with large projects
  - You're likely to complete the projects with reasonable effort.
- If you at least know some static-typed object-oriented language
  - Java, Scala, …
  - Chances are you'll need to spend (maybe significant) extra efforts though.
- If none of the above apply,
  - you'll probably have a hard time to catch up.

- This lecture's agenda:
  - An overview of Taco-DB
  - Linux programming and git basics
  - C++11 primer

Taco-DB architecture



bash basics

- `bash` is the default shell and/or available in many Linux OS
  - CentOS on CSE student servers comes with `tcsh` by default – similar but not entirely the same
  - You can switch to bash by typing "`bash`"
- A few commonly used commands:
  - list the current directory: `ls -al`
  - create a directory
    `mkdir repo` # creates a new directory "`repo`"
  - change directory
    `cd repo` # changes working directory to "`repo`"
    `cd ..` # changes working directory to the parent directory
  - copy files/directories
    `cp README.md README2.md` # copy "`README.md`" to "`README2.md`"
    `cp -r repo myrepo` # copy the dir "`repo`" and its contents to "`myrepo`"
  - move files/directories
    `mv lab0.tar.xz.2 lab0.tar.xz` # rename "`lab0.tar.xz.2`" as "`lab0.tar.xz`"
    `mv lab0.tar.xz myrepo/` # move "`lab0.tar.xz`" into "`myrepo`" directory
    `mv myrepo myrepo2` # rename "`my_repository`" as "`myrepo2`" if the later does not exist
    # otherwise, move "`my_repository`" into "`myrepo2`"
  - extract a tarball
    `tar -xf lab0.tar.xz` # extracts "`lab0.tar.xz`" to the current directory
  - run an executable
    `./setup_repo.sh` # note: executable in current directory must begin with "`./`"

I find this tutorial written by Bruce Barnett quite useful for beginners:
POSIX Shell Tutorial (grymoire.com)

Coding environment

- We recommend working on command line using a text editor
  - Integrated Development Environments (aka IDE) are not recommended
    - Use at your own risk! A list of reasons:
    - code completion is often broken in many popular IDE for C/C++
    - incompatible build system (we use `cmake`)
    - sometimes it can be hard to import external dependencies

- If you're not comfortable with vim/emacs
  - use a good GUI code editor
    - e.g., VSCode (free), Source Insight ($$$, but this is my personal favorite if available)
    - However, still *no guarantee* if the code completion works/code analysis properly.
  - Use command line for build/run/test.

Coding environment

- We recommend using your own computer for debugging
  - x86_64, bare metal or virtual machine (incl. WSL2)
  - Linux (not too old)
  - gcc/g++ (>= 7) or clang/clang++ (not tested)
  - cmake (>= 3.13)
  - pkg-config (recommend >= 0.29, older versions are very slow on first build!)
  - autoconf, make, python3
- If you don't have access to x86_64 hardware, the alternative is
  - CSE student servers (timberlake.cse.buffalo.edu or metallica.cse.buffalo.edu)
  - Your home directory is mounted on an NFS – you can use either machine
  - SSH and command line only
  - Special instructions for setups (need specific packages in `/util`)

Setting up SSH for Github and CSE student servers

- If you have never generated the SSH authentication keys,
  - it's likely you need to set up that for Github and (optionally) CSE student server access
  - check if you have `~/.ssh/id_rsa` and `~/.ssh/id_ras.pub`
- To generate a key pair

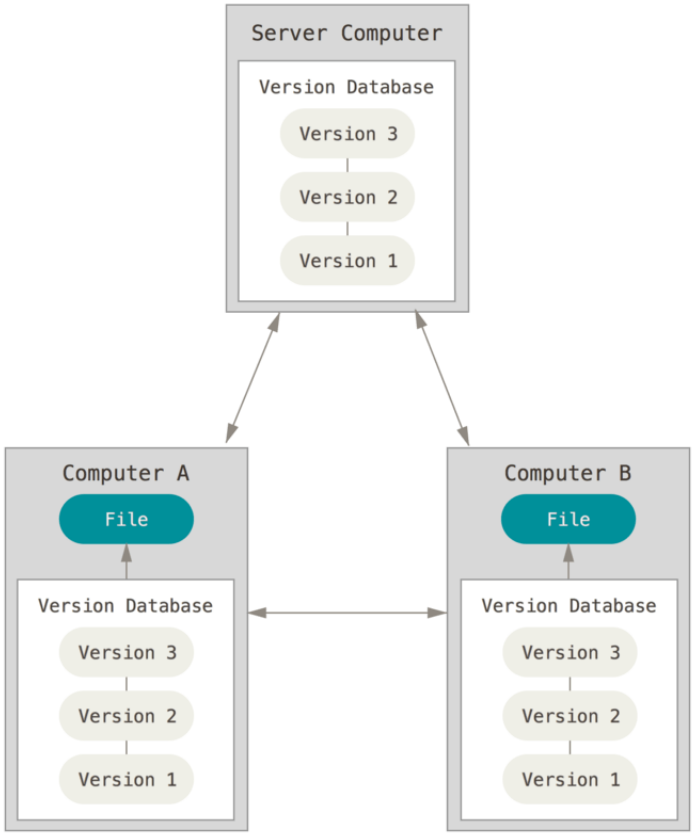
  ```
  ssh-keygen    # follow the prompts
  ```

- You should have
  - a private key `~/.ssh/id_rsa`, never share it with anyone
  - and a public key `~/.ssh/id_rsa.pub`, upload it to the severs you want to access
    - Hint: use "`cat ~/.ssh/id_rsa.pub`" to print it to your terminal for copying
    - Github -> Settings -> SSH and GPG keys
    - CSE student servers (optional)

      ```
      ssh-copy-id <your-ubitname>@timberlake.cse.buffalo.edu
      # or (the two servers store your home directory in a shared NFS)
      ssh-copy-id <your-ubitname>@metallica.cse.buffalo.edu
      ```

git basics

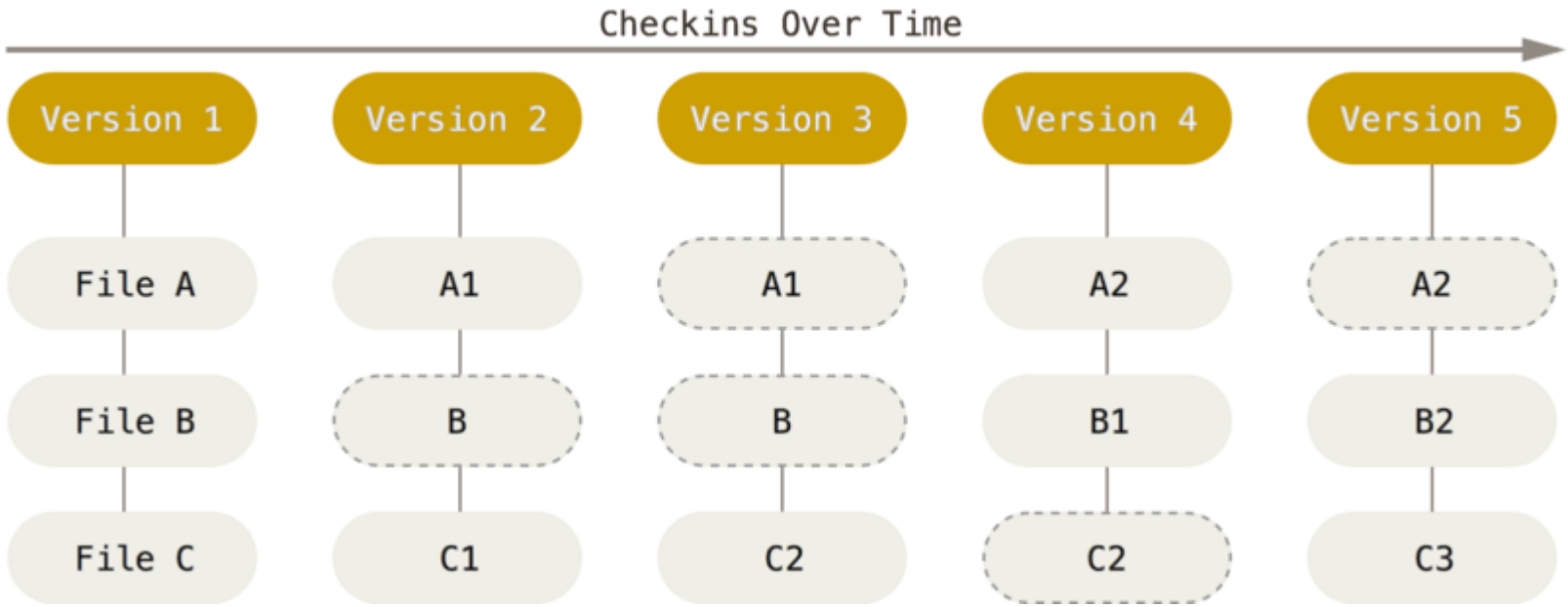
- git is a fully distributed version control software
  - Working locally on a full copy of your repository
  - Usually there's a remote server that maintains a copy for collaboration and sharing



Picture from [Git - About Version Control (git-scm.com)]

git basics

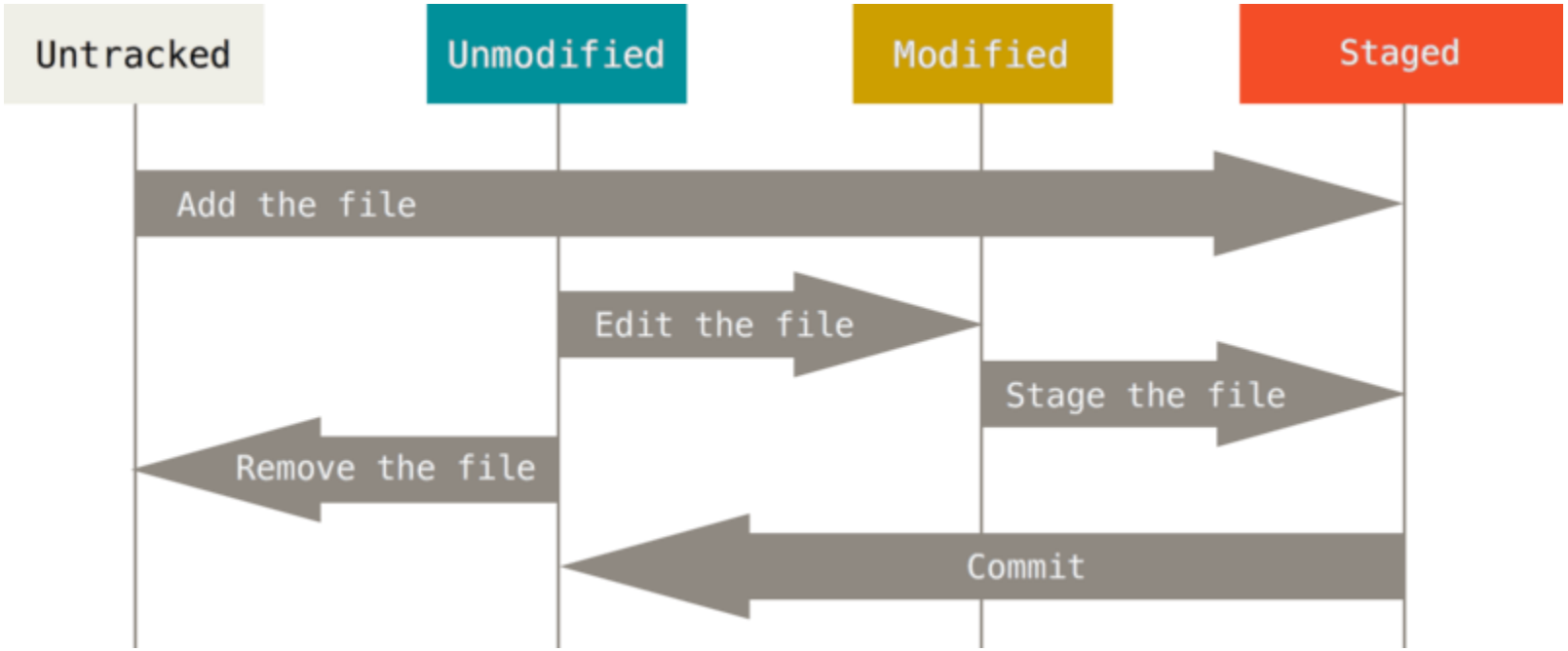
- Commits are snapshots of versioned files



Picture from [Git - What is Git? (git-scm.com)]

git basics

- Lifecycle of files in a repository



Picture from Git - Recording Changes to the Repository (git-scm.com)

git basics

- Initializing a new repository

```
mkdir myrepo && cd myrepo
git init
# import/edit your initial files
git add -A
git commit -m "initial commit"
```

git basics

- To collaborate with others, you'll need a hosting service for storing a copy
  - Github is one example, and we'll use Github for this course project
  - Once you create a repository, you may find the SSH link `git@github.com:username/repository_name.git`
  - The link allows you to read/write the remote copy of the repository hosted on Github
  - This is also the link to submit in project 1 – lab 0
    - You must add `buffalo-cse562-sp22` as a collaborator, as well as your teammate
    - *Don't add anyone else!* We log your list of collaborators if we find you have more collaborators than allowed.

git basics

- Before you do anything, configure your git:
  - Set your name
    ```
    git config --global user.name "<Your name>"
    ```
  - Set your email
    ```
    git config --global user.email "<your email>"
    ```
  - Optional but useful, set your text editor
    ```
    git config --global core.editor "<editor-of-your-choice>"
    ```

git basics

- Setups (*setup_repo.sh does everything below except pushing it to remote*)
  - Initialize a local repository
    ```
    git init
    ```
  - Track the remote repository
    ```
    git remote add origin git@github.com:<username>/<reponame>.git
    ```
  - Add git submodules (for external libraries)
    ```
    git submodule add -b <branch-name> -- <git_repo_url>
    ```
  - Initialize and/or update the submodules
    ```
    git submodule update --init --recursive
    ```
  - Adding files for staging
    ```
    git add -A
    ```
  - Commit changes
    ```
    git commit -m "your commit message"
    ```
  - Rename branch (optional)
    ```
    git branch -m main
    ```
  - Set the remote tracking branch and push it to remote
    ```
    git push -u origin main     # the remote branch name should match your local branch name
    ```

git basics

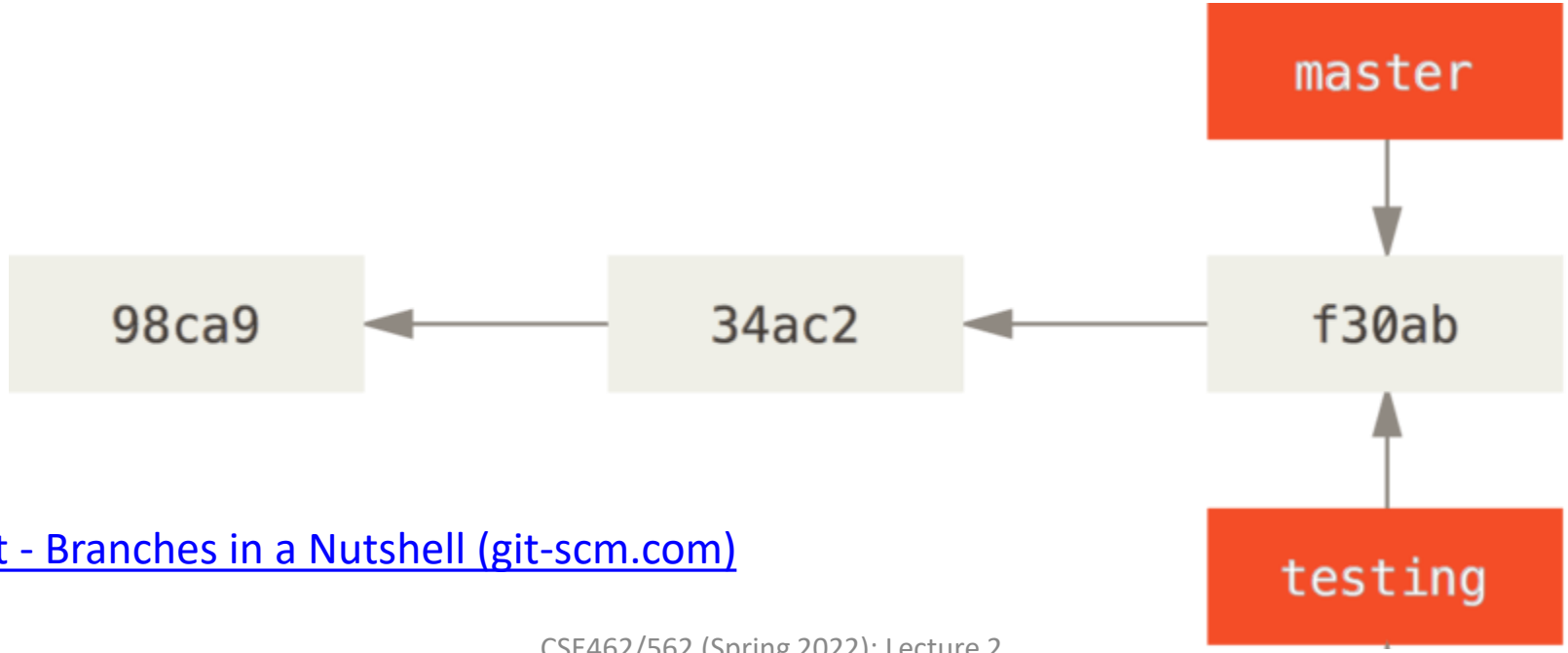
- To set up the repository on a different machine (or your teammate's)
  - Clone the repository
    # <username> is the one who created the repo, no necessarily the one who clones it
    `git clone git@github.com:<username>/<reponame>.git`
  - Initialize/update the submodules -- (only needed every time you clone it into a new directory)
    `git submodule update --init --recursive`

- Basic workflow
  - Edit/debug your source code
  - `git add -A`
  - `git commit -m "blah blah"`
  - `git push`

git basics

- Branches
  - Branches are movable pointers to commits
    - automatically moves to your latest commit
  - Usually used for tracking commits that are used for implementing some new things
  - To create a new branch

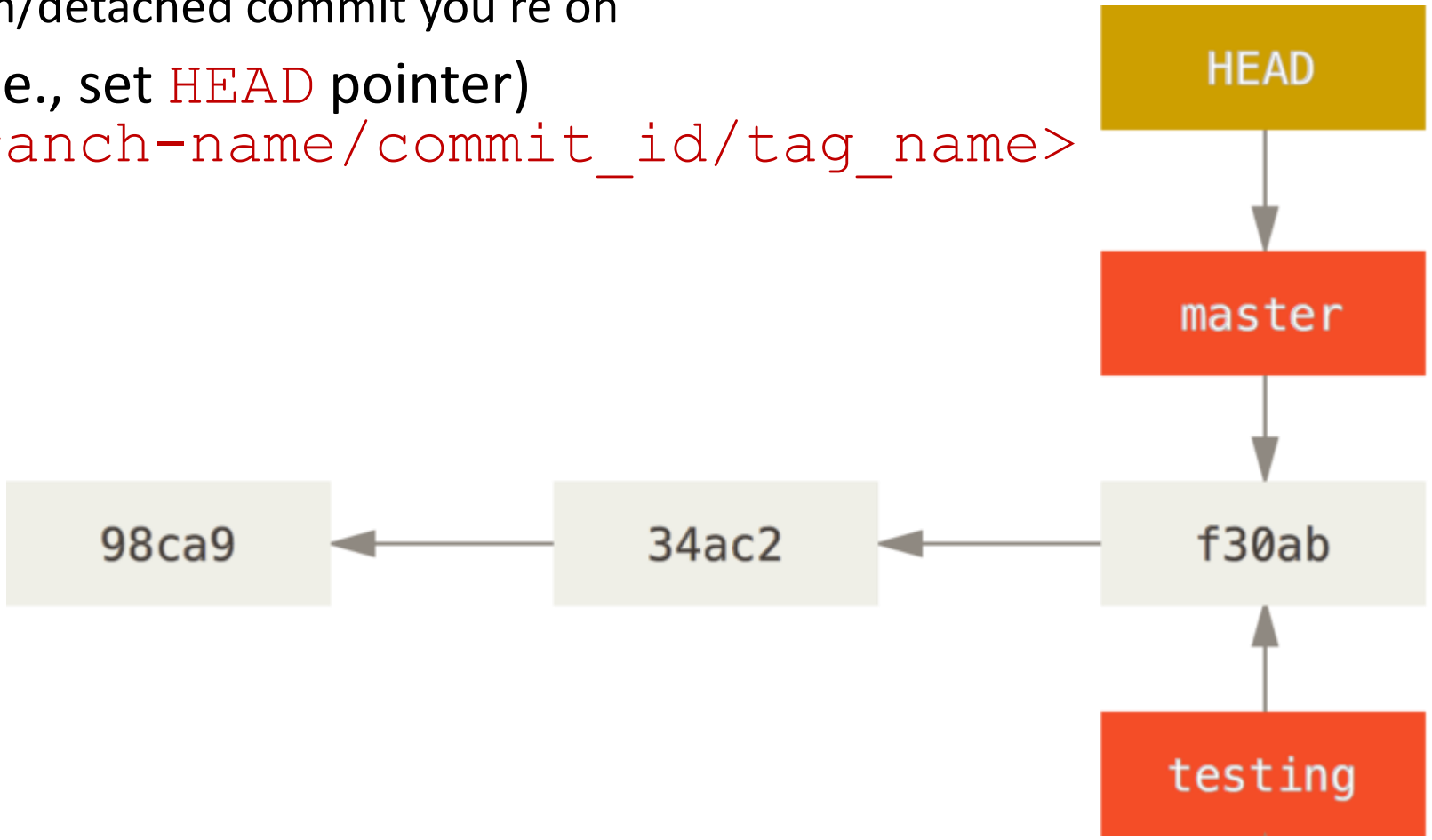
```
git branch <branch_name>
```



Picture from [Git - Branches in a Nutshell (git-scm.com)]

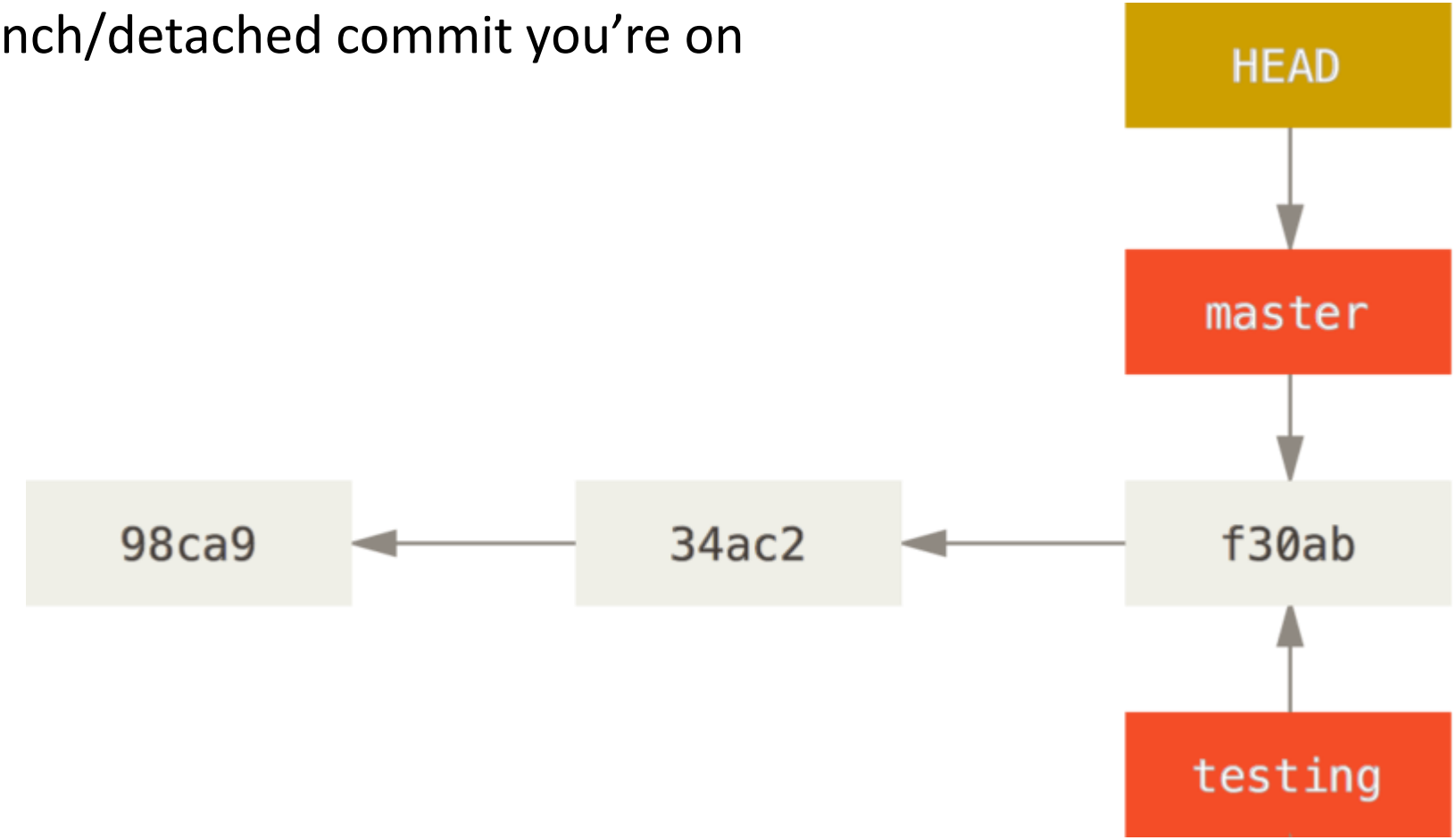
git basics

- Checkout a branch
  - git maintains a special pointer `HEAD`
  - which points to a branch/detached commit you're on
- To checkout a branch (i.e., set `HEAD` pointer)
  `git checkout <branch-name/commit_id/tag_name>`



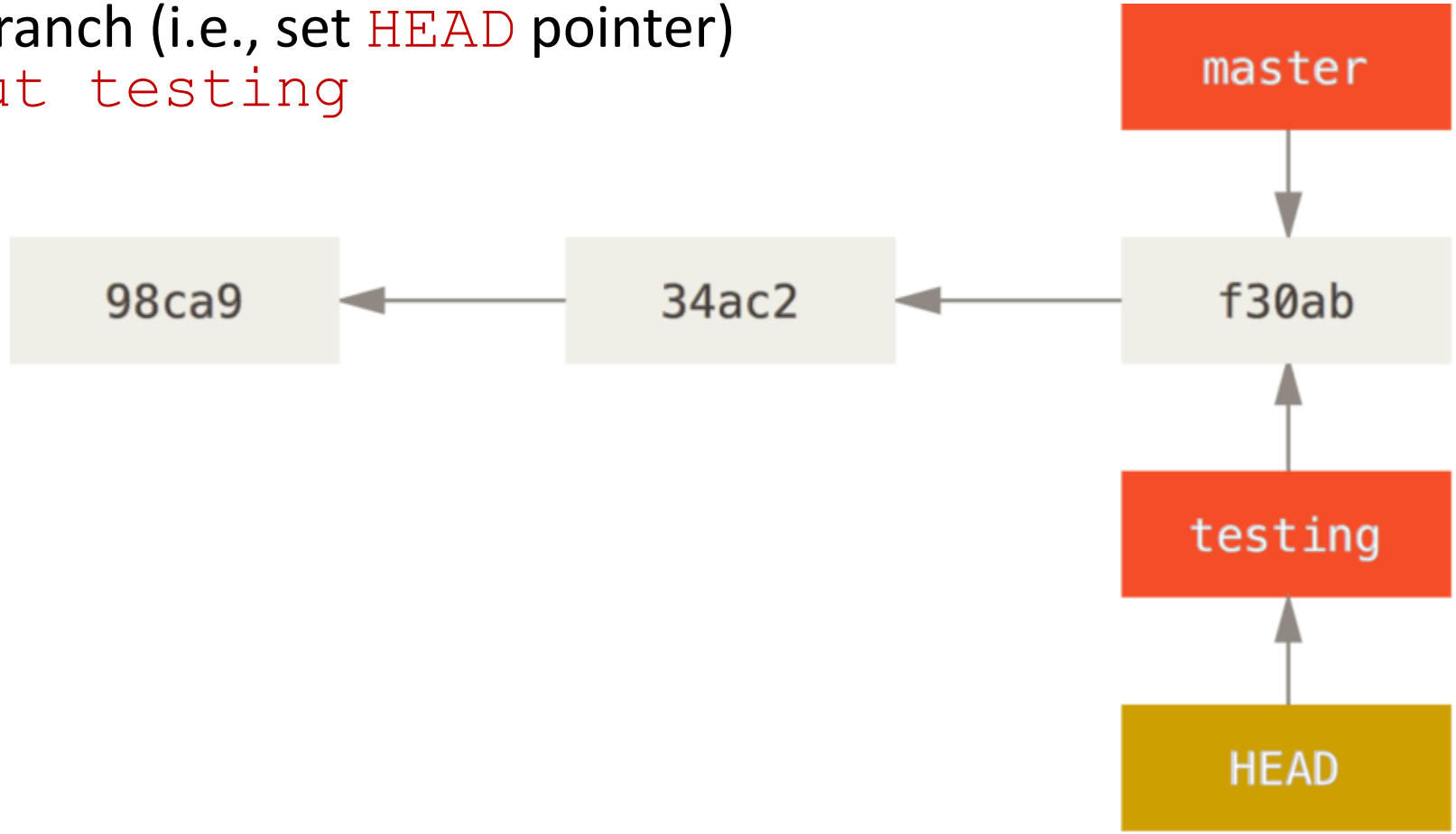
git basics

- Checkout a branch
  - git maintains a special pointer `HEAD`
  - which points to a branch/detached commit you're on



git basics

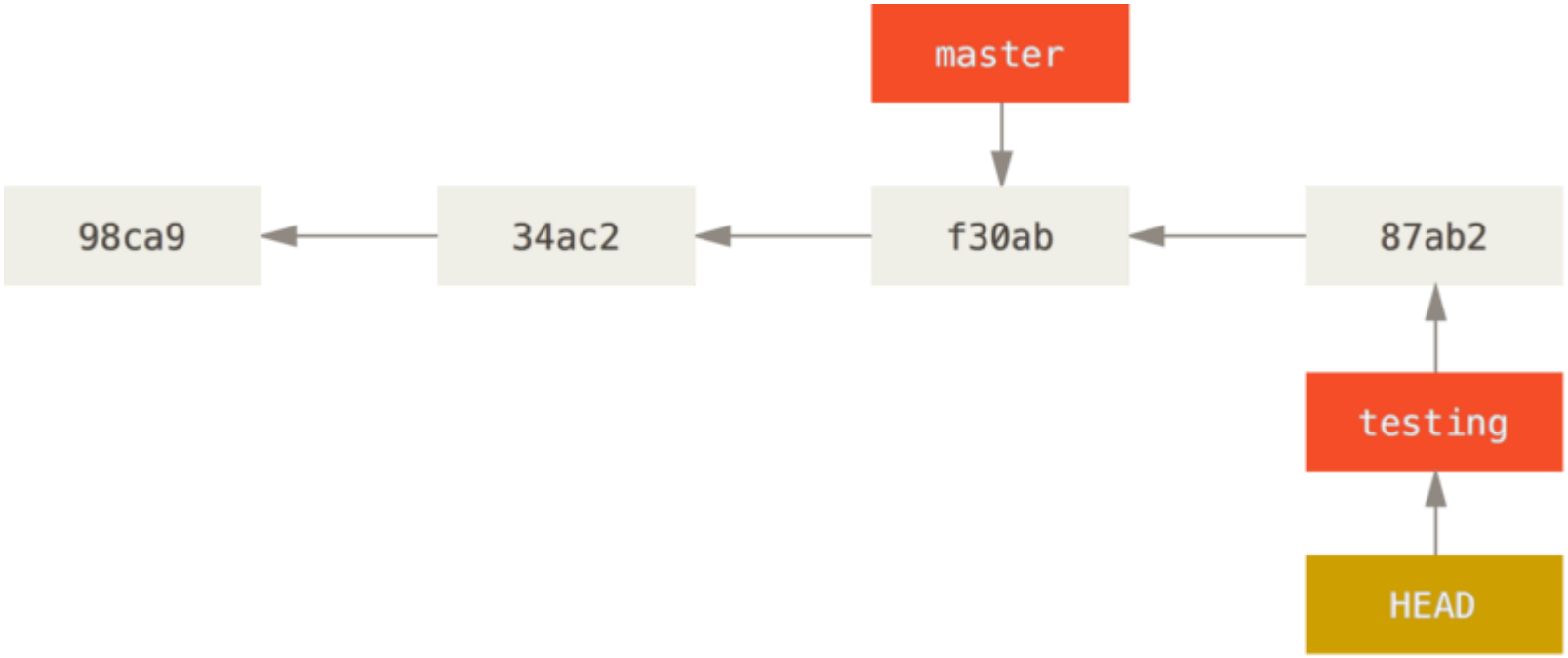
- Checkout a branch
  - git maintains a special pointer `HEAD`
  - which points to a branch/detached commit you're on
- To checkout a branch (i.e., set `HEAD` pointer)
  `git checkout testing`



git basics

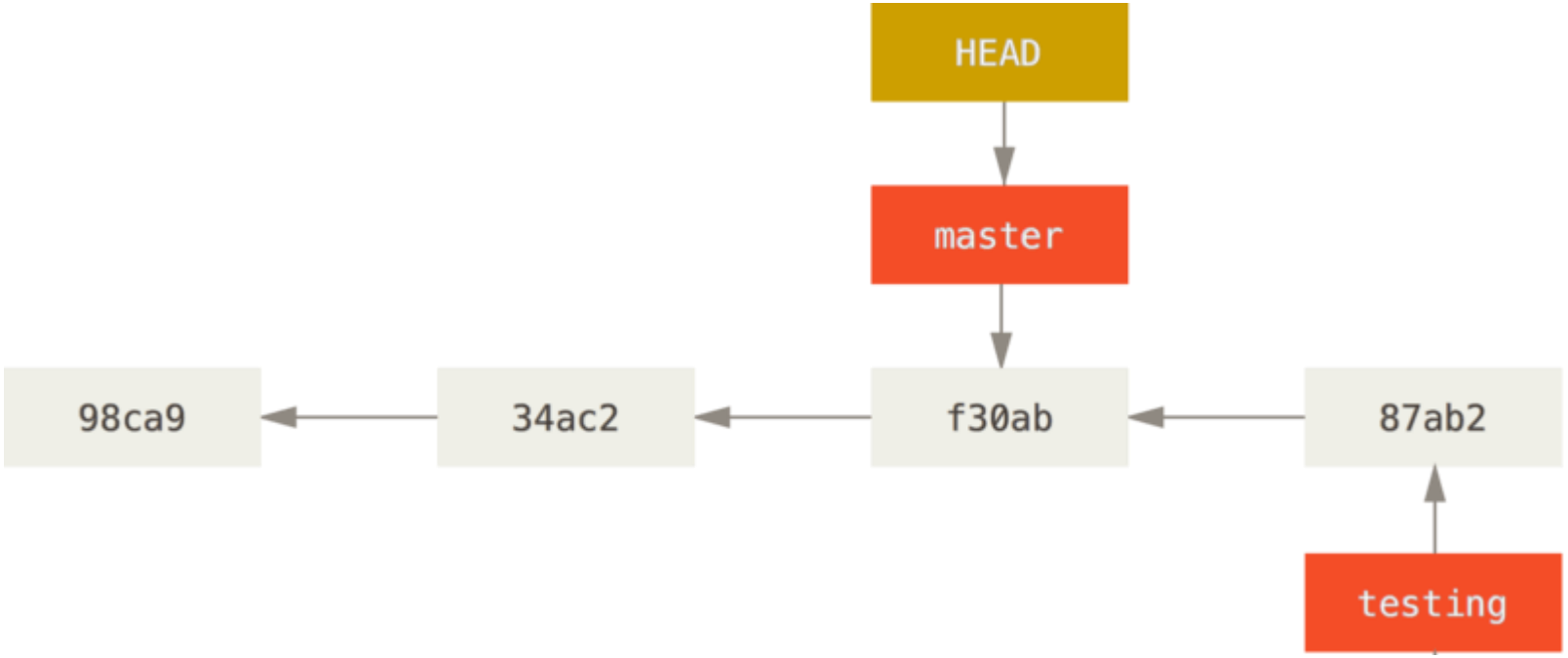
- One you commit something, the current branch and HEAD automatically get updated

```
git add -A
git commit -m "blah"
```



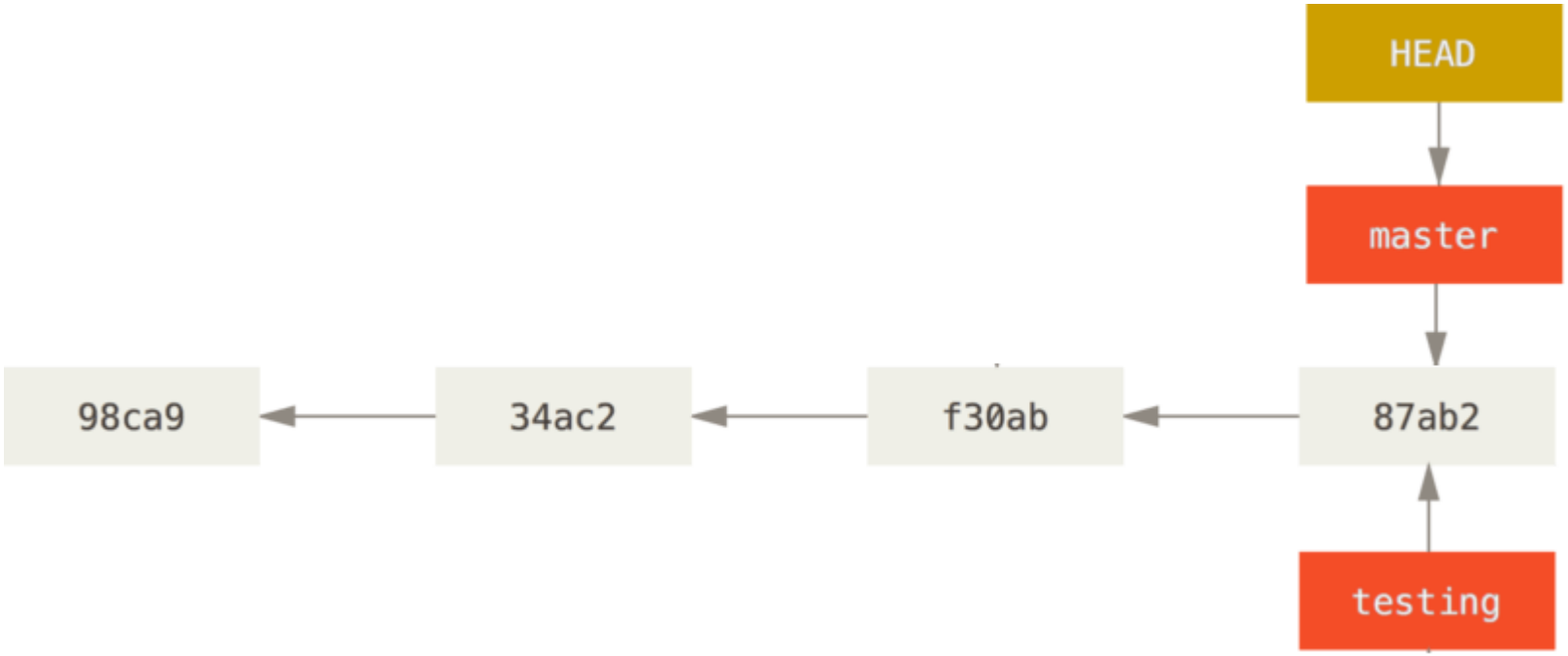
git basics

- You may go back to another branch by checking out to it
  `git checkout master`



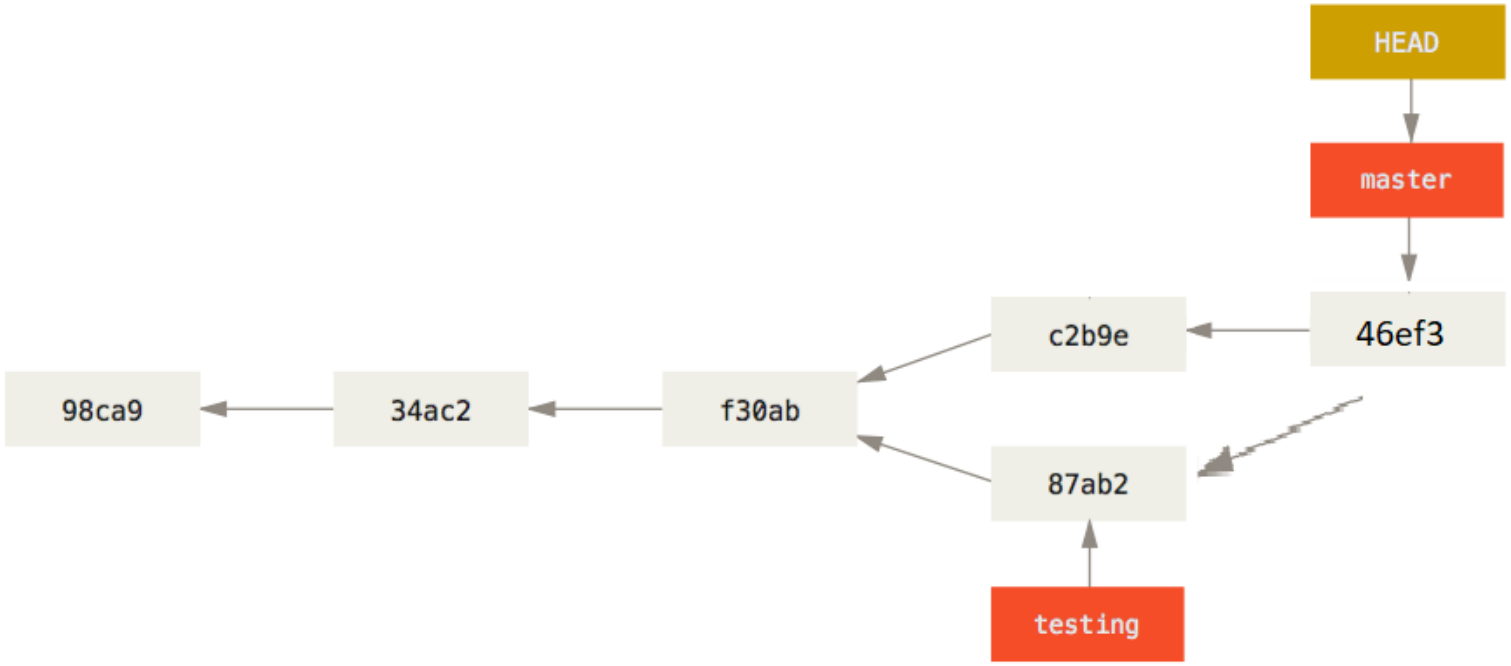
git basics

- To merge newer branches into the current branch:
  `git merge testing`
- Fast-forward merge if possible (just move the pointers)
  - Otherwise, new commit after resolving conflicts



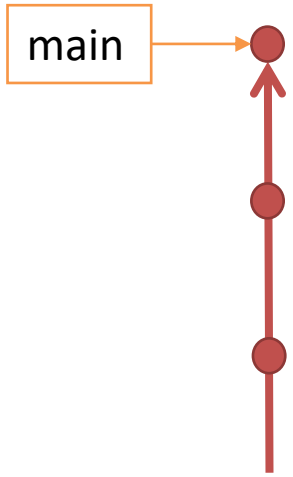
git basics

- To merge newer branches into the current branch:
  `git merge testing`
- Fast-forward merge if possible (just move the pointers)
  - Otherwise, new commit after resolving conflicts



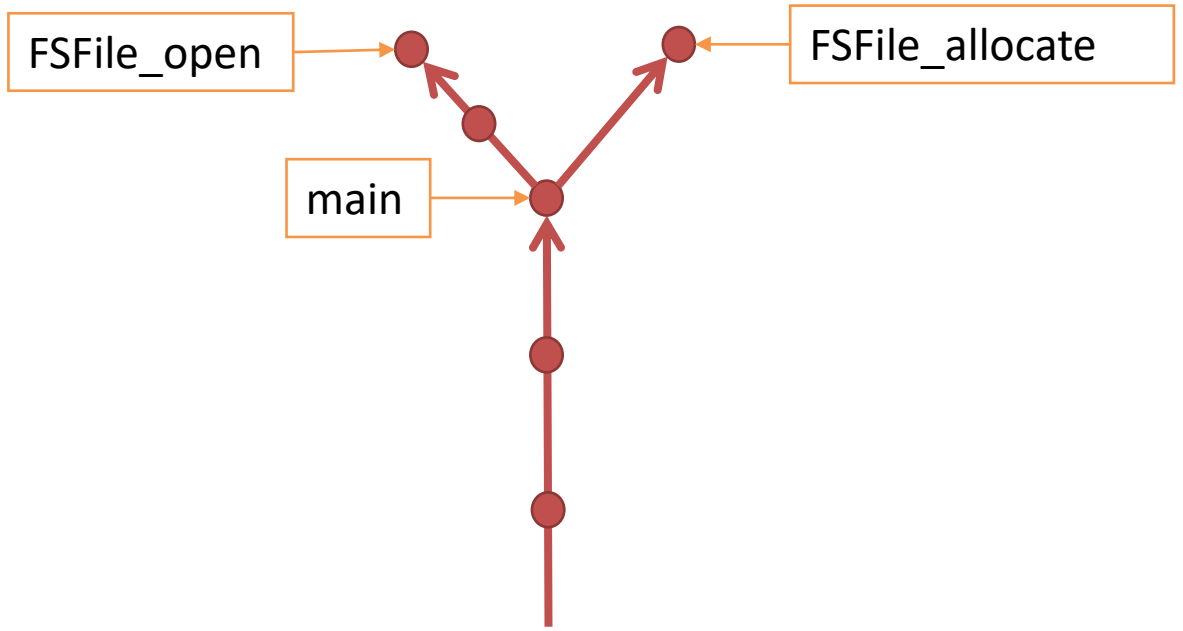
git basics

- A possible way of using branches for team collaboration



git basics

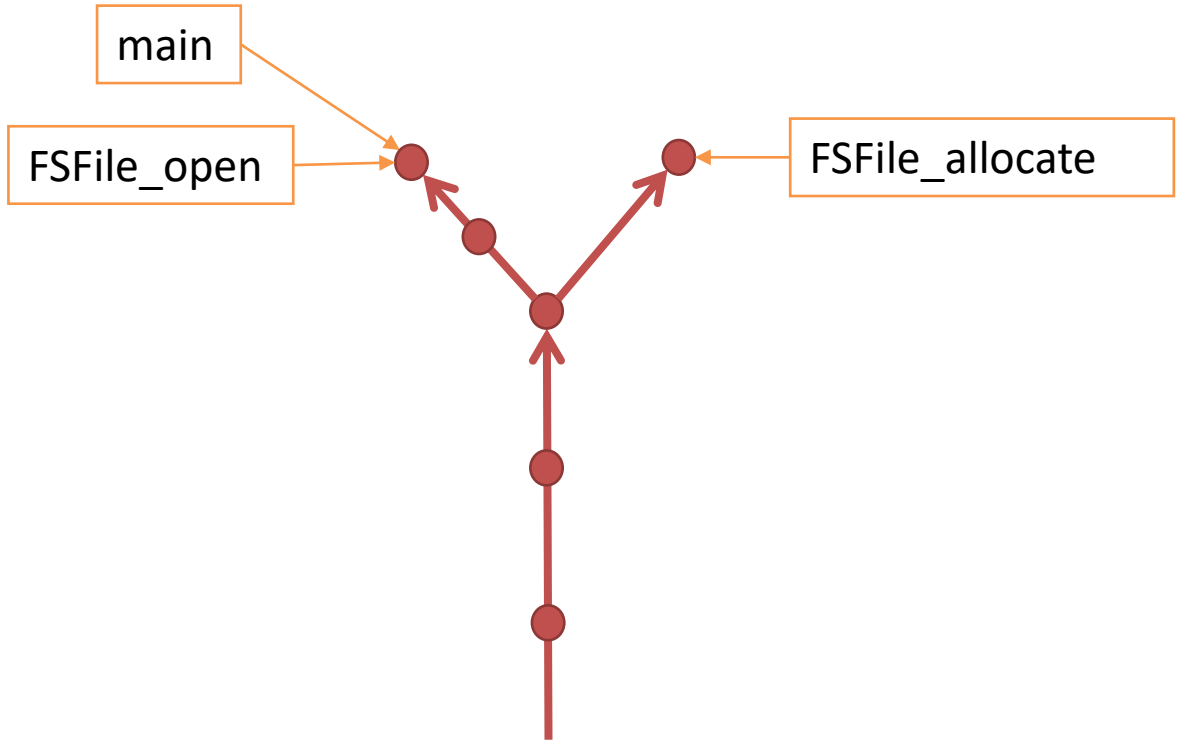
- A possible way of using branches for team collaboration
  - create a separate branch for your independent task(s) off the main branch



git basics

- A possible way of using branches for team collaboration

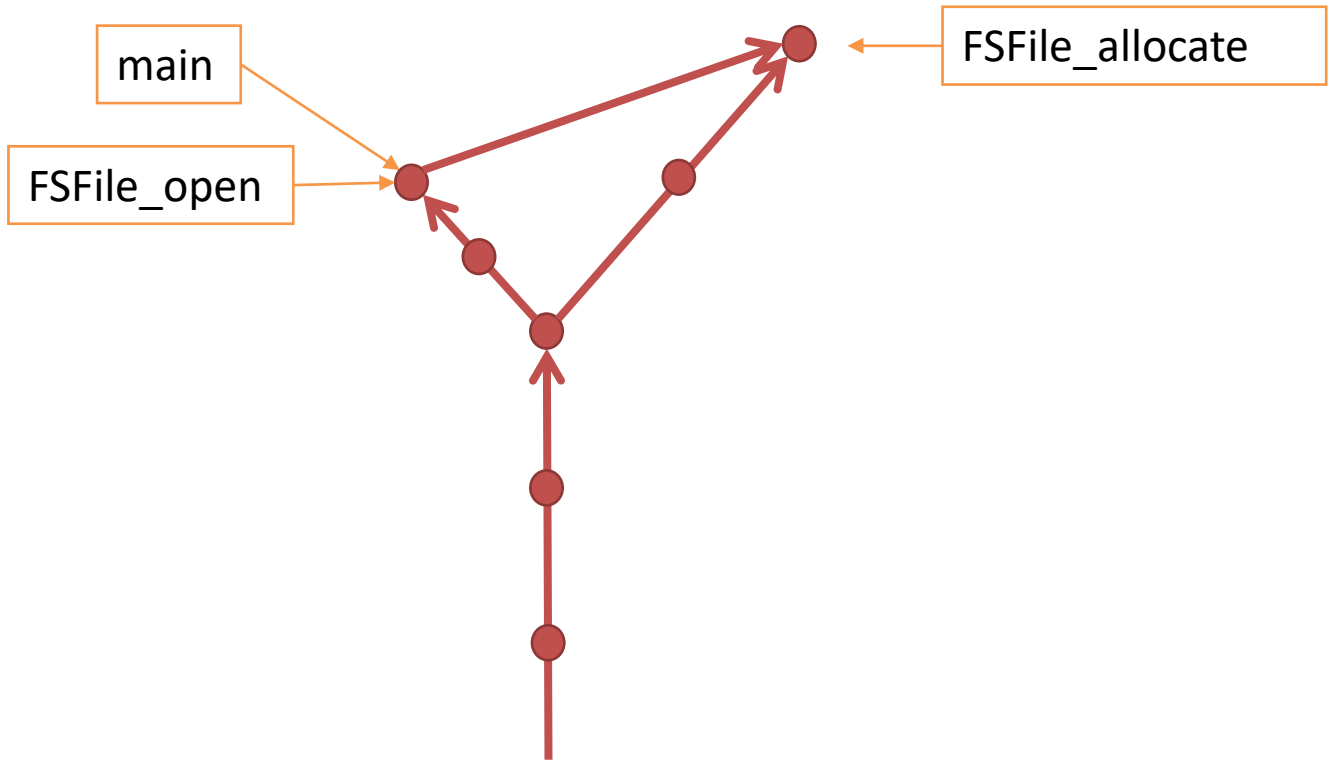
```
git checkout main
git merge FSFile_open     # fast-forward
```



git basics

- A possible way of using branches for team collaboration

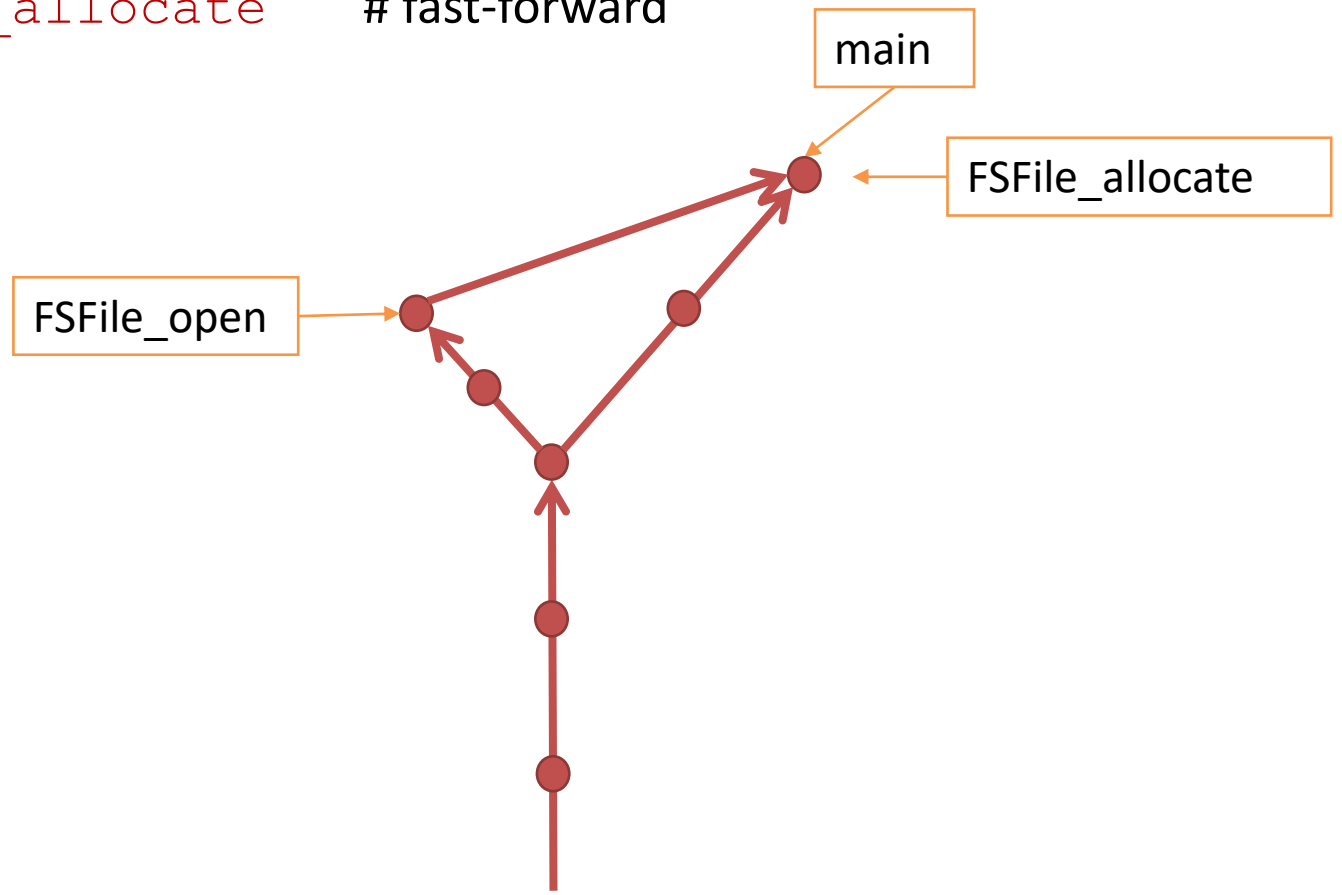
```
git checkout FSFile_allocate
git merge main   # potentially with conflicts that can't be automatically resolved!
```



git basics

- A possible way of using branches for team collaboration

```
git checkout main
git merge FSFile_allocate     # fast-forward
```



git basics

- Tags

```
git tag some-tag-name
git push --tags
```

  - A name for a specific commit, not movable
  - When checking out a tag, it is in a detached state (i.e., not on any branch)
  - The same as if you checkout the commit id

  - When you make submission, we'd recommend making a tag instead of branching
    - so that you can test your features independently
    - *You can't submit a commit ID without a branch/tag name*

- Read more at Git - Book (git-scm.com)

How to set up project repository

- For project 1, lab 0, each team should create a new Github Repository
  - Download `lab0.tar.xz` from Autolab
    - There might be a version suffix `.x (currently .2)`
  - Extract the tarball (`tar xf lab0.tar.xz`)
    - and rename the directory to your repo name
  - Run `setup_repo.sh`, which does the following
    - Initialize your local repository
    - Set up Git submodule for external libs (`jemalloc, Abeisl, GoogleTest`)
    - Make the first commit and rename the default branch to `main`
  - At this point, your local repository should work fine, check if it builds and tests ok
  - Push it to remote

How to build your code

- We use the `cmake` build system
- To create a new build directory from your repository root

```
cd myrepo
cmake -B build .
```

- To build your code

```
cd build
make
```

- -j flag does not work properly due to deps on a few code generation scripts

- For those who are using CSE student servers, follows instruction in project 1 to customize the compilers and pkg-config to use.

How to test your code

- We use `GoogleTest` with its `ctest` integration
- To run all test cases:
  `ctest -V` # -V shows GoogleTest outputs
- To run a specific test case, say `BasicTestRepoCompilesAndRuns.TestShouldAlawysSucceed`
  `ctest -R "^BasicTestRepoCompilesAndRuns.TestShouldAlwaysSucceed$" -V`
  Or, assuming you're in build directory
  `./tests/BasicTestRepoCompilesAndRuns --gtest_filter="*.TestShouldAlwaysSucceed"`

- Invoke test excutable with `--disable_logs=false` flag to enable logging output
  - doesn't work with ctest!
- Invoke test executable or ctest with `--help` for list of options

Using GDB

- GDB is a debugger that allows you to inspect a running program
- Common usages:

  `gdb ./tests/some-test-executable`

  - To set a breakpoint: `b <filename>:<lineno>/<function_name>`
  - To list breakpoints: `info br`
  - To start debugging: `r [optional command line flags]`
  - Run to next line in the current function: `n`
  - Step into a function: `s`
  - Continue execution until end or some breakpoint: `c`
  - Print a variable/expression: `p [expr]`
    - Print array: `p *some_array @ length`
  - Print stack trace: `bt`
  - Kill the current debugging process: `k`
  - Quit gdb: `q or Ctrl+D`
  - Things that might also be useful for debugging:
    - watchpoint (`w`) and auto display (`display`)
- Read more in GDB user manual

C++11 primer

- This is not meant to be a tutorial on C++!
  - Assuming you are at least familiar with the major revision of C/C++ in the 90s
    - C90/C99 and/or C++98/C++03
  - Please find a good textbook if you're not familiar with them

- C++ has many paradigms and caveats
  - The newer syntax and libraries introduced in C++11
  - The common pitfalls you might want to pay attention to
  - The design choices we made in Taco-DB

Major features of C++11

- `auto` and `decltype`
- move semantics
- rvalue references
- smart pointers
- lambda expressions
- variadic templates
- list initialization
- nullptr
- range for-loop
- …

auto and decltype

- Define variables with type deduced from placeholder specifiers

```
auto x = 1;                                  // type of x is int
auto v = new std::vector<int>(5,0);  // type of v is std::vector<int>*
const auto &pv = v;                  // type of pv is const std::vector<int>&

// type of i is std::vector<int>::iterator
for (auto i = v.begin(); i != v.end(); ++i) {…}
```

- Declare types from an entity or an expression

```
decltype(v.at(0)) i = v.at(0); // type of i is int&
auto i = v.at(0);                    // type of i is int

// iterator is an alias of std::vector<int>::iterator
typedef decltype(v.begin()) iterator;
```

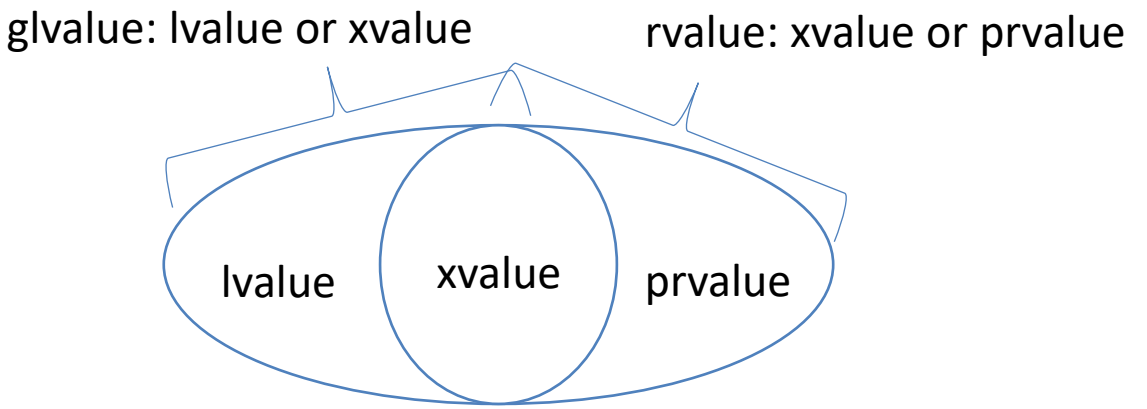
Placeholder type specifiers (since C++11) - cppreference.com
decltype specifier - cppreference.com

Move semantics and rvalue references

- Move semantics
  - Transfer the resource of an object to another
    - which puts the old object in some valid but unspecified state
    - usually means you can't use the old object any more except for explicitly specified
  - For instance, moving of an std::vector moves its internal array pointer from one to another. After the move, the old vector is not safe for dereference (in GLIBC, it's set to nullptr).

Move semantics and rvalue references

- Value categories
  - lvalue: an expr that determines the identity of an object or function
    e.g., a variable or an lvalue reference, etc.
  - prvalue: an expr that computes a built-in operator or initializes an object
    e.g., `1 + 2`, `std::vector<int>(5, 0)`, etc.
  - xvalue: an object that may be moved (i.e., resource can be reused)
    e.g., `std::move(v)`, `a[1]`, `b.x` and etc.
    where v is an std::vector, a is an array, b is a struct with member x



[Value categories - cppreference.com]

Move semantics and rvalue references

- rvalue reference `T&&`
  - May be binded to rvalues (extends its lifetime)
  - move semantics (C++11) or
    copy elision (compiler optimization or mandatory in certain cases since C++17)
- Move constructor and assignment

```
struct vector {
    // move constructor, transfers resource from a to *this
    vector(vector&& a): m_arr(a.m_arr) { a.m_arr = nullptr; }

    // move assignment, deallocates resource in *this and
    //transfers resource from a to *this
    A& operator=(vector&& a) {
       m_arr = a.m_arr; a.m_arr = nullptr; return *this;
    }

    int *m_arr;
};
```

- Converting an object to xvalue for invoking move semantics using `std::move()`

```
std::vector<int> v(5, 0);
std::vector<int> v1(std::move(v)); // invokes move constructor
std::vector<int> v2(v1);           // invokes copy constructor
v = std::move(v2);                 // invokes move assignment
v2 = v1;                           // invokes copy assignment
```

Value categories - cppreference.com

Smart pointers

- `std::unique_ptr<T>`
  - Represents ownership of an object managed through a pointer (usually heap allocated)
    - Automatically calls delete upon desctruction
  - Not copiable; only movable
  - `absl::WrapUnique()` and `absl::make_unique()`
    - Drop-in replacement for missing functions in C++11
  - Default delete invokes delete expressions (`delete` and `delete[]`)
    - We defined `unique_malloced_ptr`, which invokes `free()` in delete instead
- `std::shared_ptr<T>`
  - Represents shared ownership of an object through reference counting
  - We use this when there might be multiple ownerships of the same object (e.g., a cached item)

RAII-style objects in Taco-DB

- `taco::ResourceGuard<T>`
  - Very similar to std::unique_ptr, except that it represents ownership of objects that are not necessarily pointers (e.g., buffer frame ID)
  - Maybe replaced with std::unique_ptr with "fancy pointers", but that's a hassle

- `taco::Datum`
  - Represents a type-erased objects consisting of consecutive bytes in memory
  - i.e., values stored in the database
  - A Datum conceptually "owns" its underlying object
    - Not copiable; move only
    - In reality, it *may* store a pointer to a buffer frame that must be pinned
      - It *may* store a pointer to a temporary object that will be freed on destruction
    - However, this helps us to reason about ownership of objects for memory management
    - Also encodes nullness of an object
  - References to `Datum` are `DatumRef` and `NullableDatumRef` – copiable
    - Difference is that `DatumRef` can't encode nullness but it fits in a register (8 bytes)
    - `NullableDatumRef` is less compact (16 bytes)
    - May only be "dereferenced" when the underlying `Datum` is alive

Lambda expression

- Constructs a function closure

```
[captures](parameters) -> return_type { function body }
```

  - an anonymous function that may capture variables in its scope
  - Useful for <algorithm> library

- For instance, to sort a vector using a customized comparator:

```
void sortIntVector(std::vector<int> &v, bool ascend = true) {
    std::sort(v.begin(), v.end(),
       [&](int x, int y) -> bool {
           return ascend ? (x < y) : (x > y);
        });
}
```

Lambda expressions (since C++11) - cppreference.com

Logging and error handling in Taco-DB

- We provide a `LOG()` macro for logging and error handling
  - `LOG(level, fmt, …)`
    - Fmt and … are the arguments you would pass to printf() or absl::StrFormat()
    - e.g., `LOG(kInfo, "value of variable x is %d", x);`
  - Four log severity levels (borrowed from Abseil)
    - `kInfo`: regular log message
    - `kWarning`: warnings
    - `kError`: errors that are recoverable (doesn't corrupt database)
    - `kFatal`: fatal errors that are not recoverable, e.g., program bugs, disk failure
  - LOG() in kError or kFatal will throw an exception TDBError
    - Since we are not implementing recovery, kError and kFatal are actually the same
    - Some test case will specifically check for the thrown severity levels required in the specification

Exception safety in destructor

- Since we're using exceptions for propagating errors
  - C++11 destructors are noexcept by default!
    - Uncaught exception will cause the program to be terminated
  - So, destructors *must not* log errors!
  - If your class may have errors during destruction,
    - consider add a separate Destroy() function.
  - Some of the design in Taco-DB dictates that you can't throw error, e.g.,
    - `FSFile::~FSFile()` must close the file during destructor
      - You can't log errors if `close()` call fails
      - Log a warning instead (see `include/storage/FSFile.h`)

Error handling for constructors

- Constructors are Ok to throw errors in C++
  - However, sometimes we might still want to use factory functions for allocating and constructing a new object.
  - e.g., `FSFile::Open()` returns nullptr on failure
  - This simplifies cases when the caller expects and handles errors
    - No clumsy try-catch block and no efficiency loss
    - E.g., file manager will use it to determine how many main data files there are

Summary

- We covered the basics of the course project
  - Linux programming/coding environment
  - How to access CSE student servers
  - How to make submissions
  - C++11 primer

- Project 1 – lab 0: project sign-up due next Tuesday, 2/8, 11:59 pm EST.
- Project 1 – lab1: FSFile due in less than two weeks, 2/15, 11:59 pm EST.

- Next lecture: relational model.