**the story version (so it makes sense)**

* **your computer = a little workshop.**
* **git = a time machine** that remembers every change you make to your project.
* **github = a big shelf on the internet** where you keep a copy of your project and show it to others.
* **ssh keys = a lock and a key.** you put the **lock** (public key) on your github shelf; you keep the **key** (private key) on your computer. now you can push/pull without passwords.
* **folders & files = boxes and toys.** keep boxes tidy so others find things fast.
* **compiler (clang/g++) = the cook** who turns your ingredient files (.cpp) into a meal (a program).
* **cmake = the recipe card** that tells the cook what to build and how.
* **library = a toolbox** (your math library) that apps can use.
* **headers (.h) = labels on tools**, saying “this tool exists, here’s how to call it.”
* **linker = the friend who puts everything together**. it checks that every label (function declaration) really has a tool behind it (function definition).
* **tests (googletest) = referees** who blow the whistle if your code is wrong.
* **ci (github actions) = a robot** that rebuilds and re-tests your project automatically whenever you push changes.

keep those pictures in your head; now watch how the real pieces match.

**part 1 — git + github (time machine + internet shelf)**

**what you do (one time)**

1. **install git** and tell it your name/email (so your time machine tags changes correctly).

git --version

git config --global user.name "Your Name"

git config --global user.email "your-email@example.com"

1. **make ssh key (key + lock).**

ssh-keygen -t ed25519 -C "your-email@example.com"

eval "$(ssh-agent -s)"

ssh-add ~/.ssh/id\_ed25519

pbcopy < ~/.ssh/id\_ed25519.pub # copies the "lock"

paste that lock into **GitHub → Settings → SSH and GPG keys → New SSH key**.  
test:

ssh -T git@github.com

1. **create repo on github**, then **clone with ssh** (so you use the key, not passwords):

git clone git@github.com:<you>/<repo>.git

cd <repo>

**your daily “time machine” moves**

git status # what changed?

git add . # stage changes

git commit -m "meaningful message"

git push # send to the internet shelf

**part 2 — your tidy boxes (project structure)**

you made this (perfect!):

C++ Mastery/

├── apps/

│ └── hello/main.cpp # an app (something you run)

├── lib/

│ └── math/add.h # header = label

│ └── math/add.cpp # implementation = tool

├── tests/

│ └── test\_add.cpp # referees

├── CMakeLists.txt # recipe card

└── .github/workflows/ci.yml# robot instructions

**why this is great:**

* apps/ holds little programs that *use* your tools.
* lib/ holds the tools themselves (reusable code).
* tests/ are the referees that check the tools.
* CMakeLists.txt tells the cook (compiler) how to build things.

**part 3 — cmake (the recipe card) explained line by line**

your root **CMakeLists.txt** should look like this:

cmake\_minimum\_required(VERSION 3.16) # use cmake 3.16 or newer

project(CppMastery CXX) # project name, language = C++

set(CMAKE\_CXX\_STANDARD 20) # we want C++20 features

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON) # must be C++20

# ---- library: make a toolbox called "math" from add.cpp

add\_library(math lib/math/add.cpp)

# tell people who use "math" where to find its labels (headers)

target\_include\_directories(math PUBLIC lib/math)

# ---- app: make a program called "hello" from main.cpp

add\_executable(hello apps/hello/main.cpp)

# tell "hello" to use the toolbox "math"

target\_link\_libraries(hello PRIVATE math)

# ---- testing setup (turn on referees)

include(CTest)

enable\_testing()

# download googletest automatically (so it works on mac/linux/ci)

include(FetchContent)

FetchContent\_Declare(

googletest

URL https://github.com/google/googletest/archive/refs/tags/v1.14.0.zip

)

set(gtest\_force\_shared\_crt ON CACHE BOOL "" FORCE)

FetchContent\_MakeAvailable(googletest)

# build a test program from your test file(s)

add\_executable(unit\_tests tests/test\_add.cpp)

# link tests against your toolbox + googletest main

target\_link\_libraries(unit\_tests math GTest::gtest\_main)

# discover and register tests so "ctest" can run them

include(GoogleTest)

gtest\_discover\_tests(unit\_tests)

**how the include works (the header path confusion you hit)**

* because we wrote  
  target\_include\_directories(math PUBLIC lib/math)  
  anyone using math can include the header simply as:
* #include "add.h"

not "lib/math/add.h". (shorter and cleaner)

**part 4 — your code (tools + app + referee)**

**lib/math/add.h** (label)

#pragma once

int add(int a, int b);

**lib/math/add.cpp** (tool)

#include "add.h"

int add(int a, int b) { return a + b; }

**apps/hello/main.cpp** (app using the tool)

#include <iostream>

#include "add.h"

int main() {

std::cout << "Hello, Quant Dev!\n";

std::cout << "2 + 3 = " << add(2, 3) << "\n";

return 0;

}

**tests/test\_add.cpp** (referees)

#include <gtest/gtest.h>

#include "add.h"

TEST(AdditionTest, Positive) { EXPECT\_EQ(add(2,3), 5); }

TEST(AdditionTest, Negative) { EXPECT\_EQ(add(-2,-3), -5); }

**part 5 — build & run (talking to the cook)**

from the project root:

rm -rf build # start fresh when changing CMake big stuff

cmake -S . -B build -DBUILD\_TESTING=ON # read recipe, prepare kitchen in "build"

cmake --build build # cook!

./build/hello # eat the meal (run the app)

ctest --test-dir build --output-on-failure # let referees check

you should see:

Hello, Quant Dev!

2 + 3 = 5

[ PASSED ] 2 tests.

**part 6 — github actions (the robot)**

**what it is:** a robot on github that repeats the same steps above **every time you push**—on a fresh linux machine—so everyone knows your project works.

**file:** .github/workflows/ci.yml

name: C++ CI

on: [push, pull\_request]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4 # get your code

- name: Install build tools

run: sudo apt-get update && sudo apt-get install -y cmake g++ make

- name: Configure

run: cmake -S . -B build -DBUILD\_TESTING=ON

- name: Build

run: cmake --build build -j

- name: Run tests

run: ctest --test-dir build --output-on-failure

commit & push, then open the **Actions** tab on GitHub → watch the robot work.  
add a badge to README.md so it shows green on the front page:

![CI](https://github.com/<your-username>/<your-repo>/actions/workflows/ci.yml/badge.svg)

**part 7 — common errors you hit (what they mean + quick fix)**

1. **fatal error: 'lib/math/add.h' file not found**  
   you included with a long path. because the include dir is lib/math, just do:
2. #include "add.h"
3. **Undefined symbols for architecture arm64: "add(int,int)"**  
   means you declared the function in the header but the linker didn’t see the body.  
   fix: make sure lib/math/add.cpp exists and you link the library:
4. add\_library(math lib/math/add.cpp)
5. target\_link\_libraries(hello PRIVATE math)
6. **#include <gtest/gtest.h> not found**  
   means googletest isn’t available. use the **FetchContent** block above **and** configure with tests on:
7. cmake -S . -B build -DBUILD\_TESTING=ON

check it downloaded: ls build/\_deps/ → you should see googletest-src.

1. **changes not taking effect**  
   cmake caches stuff. do a clean:
2. rm -rf build
3. cmake -S . -B build -DBUILD\_TESTING=ON

**part 8 — tiny glossary (kid-friendly)**

* **compile**: turn each .cpp file into a machine piece (.o).
* **link**: glue all pieces into one program (hello).
* **header (.h)**: promise “this function exists.”
* **source (.cpp)**: the actual function body.
* **library**: a bunch of related functions built together (your math).
* **unit test**: a small check for one thing.
* **ci**: a robot that auto-checks your code when you push.

**part 9 — mini practice (to lock it in)**

1. **add a new tool**: int mul(int,int) in lib/math/ with tests.
   * update add.h and add.cpp (or create mul.h/.cpp)
   * add tests in tests/test\_add.cpp
   * build locally, run tests, push, confirm CI green.
2. **break a test on purpose** (e.g., expect add(2,2)==5) and watch CI go red.  
   then fix it and watch CI go green. this teaches you the feedback loop.
3. **document**: write 5 bullet points in docs/ about what you learned today.

**part 10 — where to go next (when you’re ready)**

* **sanitizers** (catch memory bugs):
  + add to CMake:
  + option(ENABLE\_SANITIZERS "Enable ASan/UBSan" OFF)
  + if(ENABLE\_SANITIZERS AND CMAKE\_CXX\_COMPILER\_ID MATCHES "Clang|GNU")
  + add\_compile\_options(-fsanitize=address,undefined -fno-omit-frame-pointer -g)
  + add\_link\_options(-fsanitize=address,undefined)
  + endif()
  + in CI, add a second job that configures with -DENABLE\_SANITIZERS=ON.
* **clang-format / clang-tidy** (style & lint).
* **docker** (prove it builds in a container).
* **matrix builds** (ubuntu + macos).

**quick command cheat-sheet**

# build & test locally

rm -rf build

cmake -S . -B build -DBUILD\_TESTING=ON

cmake --build build

./build/hello

ctest --test-dir build --output-on-failure

# git basics

git status

git add .

git commit -m "message"

git push

# copy SSH public key again (mac)

pbcopy < ~/.ssh/id\_ed25519.pub

Great question 🙌 — now that your project is set up correctly, adding new code is just a matter of **knowing where to put it** and **keeping CMake + tests in sync**. Let’s walk through it step by step, as if you’re adding a new function (say, multiply) to your project.

**🔹 1. Where Do New Codes Go?**

Think of your repo as a house with rooms:

* **lib/** → Reusable tools (libraries). Example: add, multiply, later orderbook.
* **apps/** → Small runnable programs (demos/tests that use your tools). Example: hello.
* **tests/** → Referees (unit tests that check each tool).

**🔹 2. Example: Add a multiply function**

**Step A — Header file (label)**

📂 lib/math/mul.h

#pragma once

int multiply(int a, int b);

**Step B — Implementation (tool)**

📂 lib/math/mul.cpp

#include "mul.h"

int multiply(int a, int b) {

return a \* b;

}

**Step C — Update apps/hello/main.cpp**

#include <iostream>

#include "add.h"

#include "mul.h"

int main() {

std::cout << "Hello, Quant Dev!\n";

std::cout << "2 + 3 = " << add(2, 3) << "\n";

std::cout << "4 \* 5 = " << multiply(4, 5) << "\n";

return 0;

}

**Step D — Add tests (referees)**

📂 tests/test\_mul.cpp

#include <gtest/gtest.h>

#include "mul.h"

TEST(MultiplyTest, PositiveNumbers) {

EXPECT\_EQ(multiply(3, 4), 12);

}

TEST(MultiplyTest, NegativeNumbers) {

EXPECT\_EQ(multiply(-2, 5), -10);

}

**🔹 3. Do I Need to Edit CMakeLists.txt?**

Nope ✅ — because your root CMakeLists.txt already has:

add\_library(math lib/math/add.cpp)

target\_include\_directories(math PUBLIC lib/math)

But now we need to **add mul.cpp to the math library**:

add\_library(math

lib/math/add.cpp

lib/math/mul.cpp # ✅ add new source

)

That’s it. CMake will rebuild math with both add and mul.

**🔹 4. Rebuild & Run**

rm -rf build

cmake -S . -B build -DBUILD\_TESTING=ON

cmake --build build

./build/hello

ctest --test-dir build --output-on-failure

✅ Expected:

Hello, Quant Dev!

2 + 3 = 5

4 \* 5 = 20

[ PASSED ] MultiplyTest...

**🔹 5. Git Workflow (save your changes)**

git status

git add lib/math/mul.cpp lib/math/mul.h tests/test\_mul.cpp apps/hello/main.cpp CMakeLists.txt

git commit -m "Add multiply function with tests"

git push

Now GitHub Actions CI will rerun → if green ✅, you know your new code works everywhere.

**🔹 6. General Rules for Future Code**

* **New feature/tool** → add to lib/ and update CMakeLists.txt.
* **Demo/runnable program** → add to apps/.
* **Check correctness** → always add a test in tests/.
* **Push early, push often** → CI will keep you safe.

👉 Do you want me to make you a **step-by-step checklist template** (like “1. make header, 2. make cpp, 3. update cmake, 4. add test, 5. build, 6. push”) that you can reuse every time you add future code?