

EECS 498 System Design of a Search Engine

Winter 2026

Lab 4: CMake

GSI: Christopher Davis
IA: Rickey Dong

Nicole Hamilton
[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

agenda

1. Announcements
2. Upcoming assignments
3. CMake
4. Google Test

agenda

1. Announcements
2. CMake
3. Google Test

announcements

Reminder:

- HTML Parser (04) and HashTable (05) autograder assignments are due this Sunday, 2/1
- Also, project plan and group photo ratings!
 - Start to think of practical goals
 - How many docs to crawl; how much data would that be?
 - Latency vs. index size vs. budget

agenda

1. Announcements
2. CMake
3. Google Test

introduction

Last lab we introduced CMake:

- C++ build system generator (industry standard)
- Abstraction over Makefiles
- Portable across hardware, OS, and even compilers
- Naturally handles a complex project file structure

Hand compiling ← Makefile ← CMake

`g++` ← `make` ← `cmake`

motivation – why CMake?

Makefiles:

- Not portable by default; flags, paths, toolchains differ across OSes and compilers
- Have to explicitly manage dependencies and file paths
- But, faster/simpler to set up for small codebases

CMake Benefits:

- (Typically) better IDE integration
- One build description → many backends (Makefiles, Ninja, XCode, VS, etc.)
- Dependencies and targets easier to manage
- Testing is built-in
- Easier to configure for cross-platform code
- Cheat answer: common in industry, so good to learn

basics

CMake operates on `CMakeLists.txt` files (CMLs).

In a codebase, a CML manages the directory and subdirectories in which it is placed.

A **project** is the top-level CMake object, and is defined by a CML at the root directory of your software project.

A root level `CMakeLists.txt` will always start with specifying the CMake version and declaring the project.

```
cmake_minimum_required(VERSION 3.20)
project(MyProjectName)
```

CMake targets

CMake has two main targets: **libraries** and **executables**.

Executables:

- Declared with `add_executable(Name)`
- Builds into executable program

Libraries:

- Declared with `add_library(Name)`
- Linked into executables or other libraries
- Three main types:
 - **Static:** code copied into final binary statically at link time
 - **Shared:** `.so/.dylib/.dll` used at runtime
 - **Interface:** no compiled output (e.g., only header files)

CMake hello world

Imagine we have a `main.cpp` that implements our program.

```
add_executable(my_program main.cpp)
```

Then, all we do is hit “build” (more on this later).

Suppose we also have `foo.h` and `foo.cpp` that we want to link with `main.cpp`. This is also simple:

```
add_executable(my_program  
    main.cpp  
    foo.cpp  
)
```

project structure

Reminder: CMLs manage their (sub)directories and export targets upwards.

CMLs can “link” to child CMLs in subdirectories via the `add_subdirectory(name)` macro.

Consider a potential search engine structure:

```
cmake_minimum_required(VERSION 3.20)
project(SearchEngine)

add_subdirectory(common)
add_subdirectory(index)
add_subdirectory(crawler)
add_subdirectory(ranker)
add_subdirectory(query)
add_subdirectory(frontend)
```

example

Let's build an example codebase!

It will have a non-trivial directory structure to demonstrate CMake linking.

You can follow along in the repo:

<https://github.com/eecs498w26/cmake-lab>

main branch already implemented.

blank branch has the CMakeLists blank, so you can fill them in.

root CML

Projects can also configure extra options, like versioning, a description, and the project's languages.

Let's declare our own example project:

```
cmake_minimum_required(VERSION 3.20)
project(Lab VERSION 1.0
        DESCRIPTION "CMake Lab Activity"
        LANGUAGES CXX)
```

/CMakeLists.txt

CMake example codebase

Today we will build:

- `lab_math` (INTERFACE Lib) - header-only, templates ok
- `lab_service` (STATIC Lib) - compiled library to link
- `lab_app` (EXECUTABLE) - demo app
- `lab_tests` (EXECUTABLE) - unit tests

Reminder:

Lots of different directory structures – CMake can handle them!

Today is just one potential example.

project structure

```

✓  app
  ✓  src
    C++  app.cpp
     CMakeLists.txt
  ✓  include
    ✓  lab_math/lab
      C  math.h
    ✓  lab_service/lab
      C  service.h
  ✓  src
    ✓  lab
      C++  service.cpp
       CMakeLists.txt
  ✓  tests
     CMakeLists.txt
    C++  math_test.cpp
     CMakeLists.txt
```

project structure

Reminder: CMLs manage their (sub)directories and export targets upwards.

CMLs can “link” to child CMLs in subdirectories via the `add_subdirectory(name)` macro.

Our project structure today is non-trivial, with many modules. Luckily, we don't have to hand-manage all the paths in the root CML. Instead, let's delegate to some subdirectory CMLs.

project structure

```
cmake_minimum_required (VERSION 3.20)
project (Lab VERSION 1.0
        DESCRIPTION "CMake Lab Activity"
        LANGUAGES CXX)

# Dumps compile commands into json
set (CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Configure compiler
set (CMAKE_CXX_STANDARD 20)
set (CMAKE_CXX_STANDARD_REQUIRED True)
set (CMAKE_CXX_EXTENSIONS OFF)

add_subdirectory (src) # Subdirectories here
add_subdirectory (app)
```

/CMakeLists.txt

math interface library

Let's declare the interface library for some math utilities.

The natural place to do this is in either the root CML or the `src/` CML. For simplicity, let's do it in `src`.

math interface library

```
# Header-only / templated math library
add_library(lib_math INTERFACE)

target_include_directories(lib_math
    INTERFACE
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR} /../include/lib_math>
        $<INSTALL_INTERFACE:include/lib_math>
)
```

`/src/CMakeLists.txt`

We declare it as an INTERFACE library.

We also have to specify the include path. Recall that the headers are within `/include/lib_math`; these get exported when the library is linked elsewhere.

static library

Let's use our header-only interface library to build a static library (i.e., one that does have a compiled output).

```
add_library(my_lib STATIC
    path/my_file.cpp
    path2/my_file2.cpp
)
```

To use the other library, we have to link it:

```
target_link_libraries(my_lib PUBLIC other_lib)
target_link_libraries(my_lib PRIVATE other_lib)
```

Public: exposes to consumers. Private: hides dependency.
I'll elaborate more on public vs private in a bit.

static library (example)

```
# Static library (builds compiled artifacts)
add_library(lab_service STATIC
    lab/service.cpp
)

# Service uses math types and exposes math headers to consumers
target_link_libraries(lab_service PUBLIC lab_math)

target_include_directories(lab_service
    PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR} /../include/lab_service>
        $<INSTALL_INTERFACE:include/lab_service>
)
```

/src/CMakeLists.txt

public vs private vs interface

PRIVATE:

- Only this target needs it; hides from downstream consumers.

PUBLIC:

- This target and users need it; exposes dependencies.
- E.g., this target uses a dependent type in a public API.

INTERFACE:

- Only users need it.

In our examples:

- `lab_service PUBLIC lab_math`
- `my_exec PRIVATE lab_service`

executable

Now, let's build an executable using our service (static) library. Let's put it in the **/app** subdirectory, for more practice.

```
add_executable (lab_app
    src/app.cpp
)

target_link_libraries (lab_app PRIVATE lab_service)
```

/app/CMakeLists.txt

building

Idea: don't clutter your codebase with build artifacts.

Instead, do it outside your source directory; e.g., in a build directory. This is called an *out-of-tree* build. It makes cleaning and a `.gitignore` much simpler.

Go to project root; there are two common CLI options to build:

```
└─christopherdavis ~/CMAKE-LAB/  
└─> mkdir build && cd build  
└─christopherdavis ~/CMAKE-LAB/build  
└─> cmake ..
```

```
└─christopherdavis ~/CMAKE-LAB/  
└─> cmake -S . -B build  
└─christopherdavis ~/CMAKE-LAB  
└─> cmake --build build
```

testing & 3rd party dependencies

Let's add some tests for our library.

The industry standard C++ unit test framework is **Google Test**. To use it, we will need to fetch and link it, since it's a 3rd party dependency.

But first, let's add a testing directory.

```
cmake_minimum_required (VERSION 3.20)
project (Lab VERSION 1.0
        DESCRIPTION "CMake Lab Activity"
        LANGUAGES CXX)

option (LAB_BUILD_TESTS "Build unit tests" ON)

set (CMAKE_EXPORT_COMPILE_COMMANDS ON)

add_subdirectory (src)
add_subdirectory (app)

if (LAB_BUILD_TESTS)
    include (CTest)
    enable_testing ()
    add_subdirectory (tests)
endif ()
```

/CMakeLists.txt

3rd party dependencies

```
include(FetchContent)

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG v1.14.0
)

FetchContent_MakeAvailable(googletest)
```

/tests/CMakeLists.txt

Unfortunately, C++ has no canonical package manager, unlike Cargo for Rust. Common ones are vcpkg and conan.

Luckily, we can do it just in CMake, using **FetchContent**.

unit test executables

```
add_executable (lab_tests
  math_test.cpp
)

target_link_libraries (lab_tests
  PRIVATE
    lab_math
    GTest::gtest_main
)

# CTest integration
include (GoogleTest)
gtest_discover_tests (lab_tests)
```

[/tests/CMakeLists.txt](#)

agenda

1. Announcements
2. CMake
3. Google Test

Google Test (GTest)

What is Google Test?

- Google's open source C++ unit testing framework
- Very popular in industry

It lets you write small, fast tests for functions/classes.

Test runner:

- Automatically discovers tests
- Runs them
- Reports failures with detailed file/line/error info

TEST() macro

```
TEST(SuiteName, TestName) { }
```

This defines a single unit test. Usual convention has

SuiteName define a unit, and **TestName** names a specific behavior to test.

Instead of having **assert()** or manual boolean checks, GTest has several macros to validate output.

GTest macros

Generally, there are two types:

ASSERT_*: records failures, stops test.

- Good for preconditions
- E.g., assert a pointer output is non-null before dereferencing

EXPECT_*: records failures, keeps testing.

- Should use by default for non-blocking errors
- Allows you to test multiple behaviors without just returning on the first one

GTest macros

```
EXPECT_TRUE (cond);  
EXPECT_FALSE (cond);  
EXPECT_EQ (a, b);      // uses operator==  
EXPECT_NE (a, b);      // !=  
EXPECT_LT (a, b);      // <, <=, >, >= also exist  
  
EXPECT_NEAR (x, y, 1e-9); // floating point  
  
EXPECT_STREQ (cstr1, cstr2); // C strings  
EXPECT_STRNE (cstr1, cstr2);  
  
EXPECT_THROW (fn(), std::runtime_error);  
EXPECT_NO_THROW (fn());
```

And lots more!

example

```
#include "math_utils.h"

#include <gtest/gtest.h>

TEST(MathUtils, AddTwoNumbers) {
    EXPECT_EQ(Add(2, 3), 5);
}

TEST(MathUtils, AddWithNegative) {
    EXPECT_EQ(Add(2, -5), -3);
}
```

Try writing your own test case for **Add()**.

running tests

To run tests from the command line, we use CTest or run the binary directly.

```
ctest --test-dir build --output-on-failure
ctest -V # verbose
ctest -R {regex} # regex filter
./build/tests # or run the binary directly
```

With GTest, we can list tests, only run some tests, and more.

```
./tests --gtest_list_tests
./tests --gtest_filter=MathUtils.* # only suite
./tests --gtest_filter=*Negative* # wildcard
./tests --gtest_filter=MathUtils.*:*Slow* # exclude
./tests --gtest_repeat=100 --gtest_break_on_failure
```

CMake targets

```
target_sources(MyLibrary
  PRIVATE
    library_implementation.cxx

  PUBLIC
    FILE_SET myHeaders
    TYPE HEADERS
    BASE_DIRS
      include
    FILES
      include/library_header.h
)
```