

EECS 498 System Design of a Search Engine

Winter 2026

Lab 5: GTest

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. Google Test

agenda

1. Announcements
2. Google Test

announcements

Reminder:

- **HashBlob** (06) and **Concurrent File Ops** (07) autograder assignments are due this Sunday, 2/8
- Also, group photo ratings are due on Friday!
 - Form link available on EdStem

agenda

1. Announcements
2. Google Test

introduction

- Last lab we covered CMake, a C++ build system generator.
- Some teams may find unit testing their components useful.
- Today, we will cover one of C++'s most popular unit testing frameworks, 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

how to use GTest

But how do we actually install, use, and run GTest?

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** .

how to use GTest

But how do we actually install, use, and run GTest?

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** .

Note: this also applies to any other testing framework you use.

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
```

unit test executables

```
add_executable (my_test_exec
    math_test.cpp
)

target_link_libraries(my_test_exec
    PRIVATE
        my_lib # Library to test
        GTest::gtest_main
)

# CTest integration
include(GoogleTest)
gtest_discover_tests(lab_tests)

/tests/CMakeLists.txt
```

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 upon the first failure

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(str1, str2); // C strings  
EXPECT_STRNE(str1, str2);  
  
EXPECT_THROW(fn(), std::runtime_error); // exceptions  
EXPECT_NO_THROW(fn());
```

And lots more!

example

```
#include    "math.h"

#include    <gtest/gtest.h>

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

TEST( Math ,  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
```