GoogleTest (gtest) — Step-by-step Tutorial for C++

Generated by ChatGPT (GPT-5 Thinking mini).

Table of contents

What is GoogleTest?

GoogleTest (often called gtest) is Google's C++ testing and mocking framework. It provides a rich set of assertions, test fixtures, parameterized tests, and integrates well with CMake and common CI systems. See the official user's guide for the canonical reference.

Installing GoogleTest (options)

Common installation/use options:

- Use FetchContent / add_subdirectory in CMake (recommended for reproducible builds). - Use your package manager (vcpkg, apt, brew) when available (note: some distro packages provide only headers and require building the library). - Add googletest as a git submodule and add_subdirectory.

Pick the method that fits your workflow; the Quickstart CMake guide on the official site has recommended patterns.

Quickstart with CMake (minimal example)

Here's a small CMake example that builds and runs tests. Two approaches are shown: (A) using an installed GTest (find_package) and (B) using FetchContent to include googletest in-tree.

```
Example A (find_package) - CMakeLists.txt:
---------------------------------------
cmake_minimum_required(VERSION 3.14)
project(MyProject LANGUAGES CXX)
enable_testing()
```

```
find_package(GTest REQUIRED)
add_executable(my_tests tests/foo_test.cpp)
target_link_libraries(my_tests PRIVATE GTest::gtest_main)
include(GoogleTest)
gtest_discover_tests(my_tests)
---------------------------------------
Example B (FetchContent) - top-level CMakeLists.txt (recommended for many projects):
---------------------------------------
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG release-1.13.0 # choose a stable release tag
)
FetchContent_MakeAvailable(googletest)
add_executable(my_tests tests/foo_test.cpp)
target_link_libraries(my_tests PRIVATE gtest_main)
include(GoogleTest)
gtest_discover_tests(my_tests)
---------------------------------------
```

Writing your first tests (TEST)

Create a file tests/foo_test.cpp:

#include

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

```
TEST(AdditionTest, HandlesPositiveNumbers) {
  EXPECT_EQ(add(2, 3), 5);
  ASSERT_NE(add(2, 2), 5);
}
```

Build the my_tests target and run the resulting binary. You will see test output indicating which cases passed/failed.

Assertions: EXPECT_ vs ASSERT_

EXPECT_* assertions report a failure but let the current test continue (non-fatal). ASSERT_* assertions immediately abort the current test (fatal).

Common assertions: - EXPECT_EQ, EXPECT_NE, EXPECT_TRUE, EXPECT_FALSE - ASSERT_EQ, ASSERT_NE, ASSERT_TRUE, ASSERT_FALSE - EXPECT_THROW, ASSERT_THROW, EXPECT_NO_THROW - EXPECT_NEAR (for floating-point ranges), ASSERT_DOUBLE_EQ

Use ASSERT_ when subsequent lines depend on the assertion (e.g., null pointer check before dereferencing).

Test Fixtures (TEST_F)

Fixtures let you share setup/teardown across related tests.

Example:

```
class DatabaseFixture : public ::testing::Test {
protected:
  void SetUp() override { db.connect(); }
  void TearDown() override { db.disconnect(); }
  Database db;
};
```

TEST_F(DatabaseFixture, InsertWorks) { EXPECT_TRUE(db.insert("row")); }

Parameterized Tests (TEST_P) & Typed Tests

Value-parameterized tests let you run the same test logic with different parameters.

Example:

class IsEvenTest : public ::testing::TestWithParam {};

TEST_P(IsEvenTest, ReturnsTrueForEvenNumbers) { int n = GetParam(); EXPECT_EQ(n % 2, 0); }

INSTANTIATE_TEST_SUITE_P(Evens, IsEvenTest, ::testing::Values(2, 4, 6, 8));

Typed tests are similar and helpful to run tests across a set of types.

Google Mock (gMock) basics

gMock is included alongside googletest and provides facilities to create mock classes and set expectations on calls.

Example interface and mock:

struct IFoo { virtual ~IFoo() = default; virtual int Do(int x) = 0; };

```
class MockFoo : public IFoo {
public:
  MOCK_METHOD(int, Do, (int x), (override));
};
```

```
TEST(UsesMock, CallsDo) {
  MockFoo mock;
  EXPECT_CALL(mock, Do(::testing::Gt(0))).WillOnce(::testing::Return(42));
  // pass `mock` to code under test and verify behavior
}
```

Running tests and command-line flags

Common flags for a gtest binary: --gtest_filter=TestsToRun (e.g., MySuite.* or -excluded)
--gtest_list_tests (lists tests without running them) --gtest_output=xml:report.xml (produce XML for CI)
--gtest_shuffle (shuffle test execution order) --gtest_repeat=N (repeat test run N times)

You can use CTest with gtest_discover_tests for seamless CI integration.

Integrating with CI and CTest

Use CMake's gtest_discover_tests or add_test() to register tests with CTest. In CI (GitHub Actions / GitLab CI), configure a job to build and run the tests; capture XML output for test reporting.

Best practices and tips

```
• Keep tests small and deterministic.
• Prefer EXPECT_* unless a failure should stop the test.
• Use fixtures for expensive setup/teardown.
• Use mocks to isolate interactions, not to test logic inside collaborators.
• Run tests frequently and include them in CI.
```

Example project layout & CMake files

Suggested layout:

```
project/
  CMakeLists.txt
  src/
    foo.cpp
  include/
    foo.h
  tests/
    CMakeLists.txt
    foo_test.cpp
```

```
tests/CMakeLists.txt example:
-------------------------------
add_executable(my_tests foo_test.cpp)
target_link_libraries(my_tests PRIVATE gtest_main)
include(GoogleTest)
gtest_discover_tests(my_tests)
-------------------------------
```

Appendix: Common macros & matchers

Short reference:

```
TEST(TestSuite, Name) - basic test
TEST_F(FixtureName, Name) - test using fixture
TEST_P/ INSTANTIATE_TEST_SUITE_P - parameterized
ASSERT_* vs EXPECT_ - fatal vs non-fatal
MOCK_METHOD(ret, name, (args), (modifiers)) - define a mock
Matchers: ::testing::Eq, Ne, Lt, Gt, Ge, Le, _ (wildcard)
```