Gorgo Continuing C++

April 11, 2026

Contents

Appendix B: Testing

How do you know your code works? You run it and check the output. But as your program grows, manually checking every feature after every change becomes impossible. **Automated tests** solve this: small programs that exercise your code and verify the results. A good test suite catches bugs before your users do and gives you confidence to refactor without fear. This appendix covers unit testing concepts, two popular frameworks (Google Test and Catch2), test-driven development, and mocking.

Unit Testing Concepts

A **unit test** tests one small piece of code — a function, a class method, or a small module — in isolation.

Arrange, Act, Assert

Most unit tests follow the **Arrange-Act-Assert** pattern:

1. **Arrange** — set up the test data and dependencies.
2. **Act** — call the function or method under test.
3. **Assert** — check that the result matches what you expect.

```
// Arrange
std::vector<int> scores = {90, 85, 92, 88};

// Act
int sum = std::accumulate(scores.begin(), scores.end(), 0);

// Assert
assert(sum == 355);
```

What Makes a Good Test

- **Focused**: tests one thing. If it fails, you know exactly what broke.
- **Independent**: does not depend on other tests or shared state.
- **Fast**: runs in milliseconds, not seconds.
- **Deterministic**: produces the same result every time.

Google Test

Google Test (also called gtest) is the most widely used C++ testing framework.

Setting Up

With CMake (Appendix A):

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

add_executable(tests test_math.cpp)
target_link_libraries(tests PRIVATE gtest_main)
```

`gtest_main` provides a `main()` function, so your test file only needs test cases.

Writing Tests

```cpp
#include <gtest/gtest.h>

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

TEST(AddTest, PositiveNumbers)
{
    EXPECT_EQ(add(2, 3), 5);
}

TEST(AddTest, NegativeNumbers)
{
    EXPECT_EQ(add(-1, -2), -3);
}

TEST(AddTest, Zero)
{
    EXPECT_EQ(add(0, 0), 0);
    EXPECT_EQ(add(5, 0), 5);
}
```

`TEST(TestSuite, TestName)` defines a test case. The suite name groups related tests.

Assertions

Macro	Checks	On failure
`EXPECT_EQ(a, b)`	`a == b`	Continues
`EXPECT_NE(a, b)`	`a != b`	Continues
`EXPECT_LT(a, b)`	`a < b`	Continues
`EXPECT_GT(a, b)`	`a > b`	Continues
`EXPECT_TRUE(cond)`	`cond` is true	Continues
`EXPECT_FALSE(cond)`	`cond` is false	Continues
`EXPECT_THROW(expr, type)`	`expr` throws `type`	Continues
`ASSERT_EQ(a, b)`	`a == b`	Stops test

`EXPECT_*` macros report failures but continue the test. `ASSERT_*` macros abort the test immediately — use them when continuing would cause a crash (e.g., after a null check).

Fixtures

When multiple tests need the same setup, use a **fixture**:

```cpp
class PlaylistTest : public ::testing::Test {
protected:
    void SetUp() override
    {
        playlist.push_back("Hey Ya!");
        playlist.push_back("Toxic");
        playlist.push_back("Crazy");
    }

    std::vector<std::string> playlist;
};
```

```
TEST_F(PlaylistTest, HasThreeSongs)
{
    EXPECT_EQ(playlist.size(), 3);
}

TEST_F(PlaylistTest, ContainsToxic)
{
    auto it = std::find(playlist.begin(), playlist.end(), "Toxic");
    EXPECT_NE(it, playlist.end());
}
```

`TEST_F` uses the fixture class. `SetUp()` runs before each test; `TearDown()` (optional) runs after each test. Each test gets a fresh instance — tests do not share state.

Running Tests

```
./tests                  # run all tests
./tests --gtest_filter="AddTest.*"  # run only AddTest suite
./tests --gtest_list_tests          # list all tests
```

Catch2

Catch2 is a header-friendly alternative to Google Test with a more concise syntax.

Setting Up

```
FetchContent_Declare(
    Catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v3.5.2
)
FetchContent_MakeAvailable(Catch2)

add_executable(tests test_math.cpp)
target_link_libraries(tests PRIVATE Catch2::Catch2WithMain)
```

Writing Tests

```
#include <catch2/catch_test_macros.hpp>

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

TEST_CASE("add returns correct sums", "[add]")
{
    REQUIRE(add(2, 3) == 5);
    REQUIRE(add(-1, -2) == -3);
    REQUIRE(add(0, 0) == 0);
}
```

Sections

Catch2's **sections** replace fixtures for many use cases. Each section runs with a fresh state:

```
TEST_CASE("Playlist operations", "[playlist]")
{
```

```
    std::vector<std::string> playlist = {"Hey Ya!", "Toxic"};

    SECTION("adding a song increases size")
    {
        playlist.push_back("Crazy");
        REQUIRE(playlist.size() == 3);
    }

    SECTION("clearing removes all songs")
    {
        playlist.clear();
        REQUIRE(playlist.empty());
    }
}
```

Each `SECTION` runs independently — the `playlist` vector is reset between sections.

Assertions

Catch2	Behavior
`REQUIRE(expr)`	Fatal — stops test on failure
`CHECK(expr)`	Non-fatal — reports but continues
`REQUIRE_THROWS_AS(expr, type)`	Checks that `expr` throws `type`

Google Test vs. Catch2

Feature	Google Test	Catch2
Syntax	`TEST()`, `EXPECT_*`, `ASSERT_*`	`TEST_CASE`, `REQUIRE`, `CHECK`
Fixtures	Class-based (`TEST_F`)	Section-based
Setup	Requires linking	Header-friendly
Maturity	Industry standard	Popular, modern
Mocking	Built-in (Google Mock)	Separate libraries

Both are excellent choices. Google Test is more common in industry; Catch2 is often preferred for smaller projects.

Test-Driven Development

Test-Driven Development (TDD) flips the usual workflow: you write the test *first*, then write the code to make it pass.

The Red-Green-Refactor Cycle

1. **Red**: Write a test for a feature that does not exist yet. Run it — it fails (red).
2. **Green**: Write the simplest code that makes the test pass (green).
3. **Refactor**: Clean up the code while keeping the tests green.

Repeat for each new feature or behavior.

Example

Step 1 — Red:

```
TEST(Factorial, BaseCase)
{
    EXPECT_EQ(factorial(0), 1);
    EXPECT_EQ(factorial(1), 1);
}
```

This fails because `factorial` does not exist.

Step 2 — Green:

```
int factorial(int n)
{
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Tests pass.

Step 3 — Refactor: The code is already clean, so nothing to change. Add the next test:

```
TEST(Factorial, LargerValues)
{
    EXPECT_EQ(factorial(5), 120);
    EXPECT_EQ(factorial(10), 3628800);
}
```

When TDD Helps

- Well-defined requirements with clear inputs and outputs.
- Bug fixes: write a test that reproduces the bug, then fix the code.
- Library code where the API is designed upfront.

When TDD is Less Useful

- Exploratory or prototype code where requirements are unclear.
- UI code or systems with heavy external dependencies.
- Performance-sensitive code where the algorithm may change drastically.

Tip: Even if you do not follow strict TDD, writing tests alongside your code (rather than after) catches bugs earlier and keeps your code testable.

Mocking

A **mock** is a fake object that replaces a real dependency in a test. Mocking lets you test a class in isolation without needing a database, network, or filesystem.

Google Mock Basics

Google Mock comes with Google Test. It uses macros to create mock classes from interfaces (abstract classes, Chapter 1):

```
#include <gmock/gmock.h>
#include <gtest/gtest.h>
```

```cpp
// Interface
class Database {
public:
    virtual ~Database() = default;
    virtual std::string lookup(int id) = 0;
    virtual void save(int id, const std::string& data) = 0;
};

// Mock
class MockDatabase : public Database {
public:
    MOCK_METHOD(std::string, lookup, (int id), (override));
    MOCK_METHOD(void, save, (int id, const std::string& data), (override));
};
```

Using the Mock

```cpp
// The class under test
class MusicService {
public:
    MusicService(Database& db) : db_(db) {}

    std::string get_track_name(int id)
    {
        return db_.lookup(id);
    }

private:
    Database& db_;
};

// The test
TEST(MusicServiceTest, ReturnsTrackName)
{
    MockDatabase mock_db;
    EXPECT_CALL(mock_db, lookup(1))
        .WillOnce(::testing::Return("Bohemian Like You"));

    MusicService service(mock_db);
    EXPECT_EQ(service.get_track_name(1), "Bohemian Like You");
}
```

`EXPECT_CALL` sets an expectation: when `lookup(1)` is called, return `"Bohemian Like You"`. If the method is never called, or called with different arguments, the test fails.

Key Google Mock Features

Macro/Function	Purpose
`MOCK_METHOD(return, name, (args), (qualifiers))`	Define a mock method
`EXPECT_CALL(mock, method(args))`	Set an expectation
`.WillOnce(Return(value))`	Return a value once
`.WillRepeatedly(Return(value))`	Return a value every time

Macro/Function	Purpose
`.Times(n)`	Expect exactly n calls

Tip: Mocking works best when your code depends on **interfaces** (abstract classes) rather than concrete classes. This is another reason to use polymorphism (Chapter 1) in your designs.

Key Points

- **Unit tests** test small pieces of code in isolation using Arrange-Act-Assert.
- Good tests are focused, independent, fast, and deterministic.
- **Google Test**: `TEST()` for test cases, `EXPECT_*/ASSERT_*` for assertions, `TEST_F` for fixtures.
- **Catch2**: `TEST_CASE` for test cases, `REQUIRE/CHECK` for assertions, `SECTION` for shared setup.
- **TDD** (Test-Driven Development) follows the red-green-refactor cycle: write the test first, then the code.
- **Mocking** replaces real dependencies with fake objects for isolated testing. Google Mock provides `MOCK_METHOD` and `EXPECT_CALL`.
- Mocking works best with interfaces (abstract classes).
- Write tests alongside your code, not after. Test coverage gives you confidence to refactor.

Exercises

1. **Think about it:** Why should tests be independent of each other? What problems can arise when tests share state?
2. **Write a test** (using Google Test syntax) for a function `bool is_palindrome(const std::string& s)` that checks if a string reads the same forwards and backwards. Include tests for "racecar" (true), "hello" (false), and "" (true).
3. **Think about it:** What is the difference between `EXPECT_EQ` and `ASSERT_EQ` in Google Test? When would you choose one over the other?
4. **Where is the bug?**

   ```
   TEST_F(PlaylistTest, CanRemoveSong)
   {
       playlist.erase(playlist.begin());
       ASSERT_EQ(playlist.size(), 2);
       EXPECT_EQ(playlist[0], "Toxic");
   }
   ```

   (Hint: what if the `SetUp` adds songs in a different order than expected?)
5. **Think about it:** When is TDD most useful? When is it less useful? Give an example of each.
6. **Write a Catch2 test** for a function `int clamp(int value, int lo, int hi)` that restricts a value to a range. Use sections to test: value below range, value in range, and value above range.
7. **Think about it:** Why does mocking require interfaces (abstract classes)? What happens if you try to mock a class with no virtual functions?
8. **What does this test check?**

   ```
   TEST(StringTest, EmptyString)
   {
       std::string s;
       EXPECT_TRUE(s.empty());
       EXPECT_EQ(s.size(), 0);
   ```

```
    EXPECT_EQ(s, "");
}
```

9. **Think about it:** The text says "write tests alongside your code, not after." Why is writing tests after the code is finished less effective?

10. **Write a mock** (using Google Mock syntax) for this interface and a test that uses it:

    ```
    class Logger {
    public:
        virtual ~Logger() = default;
        virtual void log(const std::string& message) = 0;
        virtual int message_count() const = 0;
    };
    ```

    Write a class `App` that takes a `Logger&` and has a `run()` method that calls `log("Starting")`. Test that `run()` calls `log` exactly once with the message "Starting".

Index