# Introduction

GoogleTest is C++ testing and mocking framework developed and maintained by Google. It supports many kinds of tests, not just unit tests.

Links:

* Github: <https://github.com/google/googletest>
* Guideline: <http://google.github.io/googletest/>

The framework comes with following features:

* **Various OSs**: Supportsn Linux, Windows, Mac, etc.
* **xUnit test framework**: Based on the [**xUnit**](https://en.wikipedia.org/wiki/XUnit) **testing framework**, a popular architecture for unit testing. You’ll feel right at home if you’ve used JUnit or PyUnit before.
* **Test discovery**: Automatically discovers and runs your tests, eliminating the need to manually register your tests.
* **Rich set of assertions**: Provides a variety of assertions, such as equality, inequality, exceptions, and more, making it easy to test your code.
* **User-defined assertions**: You can define your own assertions, making it simple to write tests that are specific to your code.
* **Death tests**: Supports death tests, which verify that your code exits in a certain way, making it useful for testing error-handling code.
* **Fatal and non-fatal failures**: You can specify whether a test failure should be treated as fatal or non-fatal, allowing tests to continue running even if a failure occurs.
* **Value-parameterized tests**: Supports value-parameterized tests, which run multiple times with different input values, making it useful for testing functions that take different inputs.
* **Type-parameterized tests**: Supports type-parameterized tests, which run with different data types, making it useful for testing functions that work with different data types
* **Various options for running tests**: Provides many options for running tests including running individual tests, running tests in a specific order and running tests in parallel.

# Configuration

1. Create a test directory. For example, a directory named "ut" inside your project.

myproject

├───src # This is our main source code (call code-under-test)

│ hello.cpp

│ hello.h

└───ut # This is unittest code

CMakeLists.txt

hello\_test.cpp

2. In the test directory, create a CMakeLists.txt with following content:

cmake\_minimum\_required(VERSION 3.14)

project(myproject\_uts)                     # Set your project name

# Specify compiler version, etc.

set(CMAKE\_CXX\_STANDARD 14)                  # GoogleTest requires at least C++14

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

# Clone gtest from github

include(FetchContent)

FetchContent\_Declare(

  googletest

  URL https://github.com/google/googletest/archive/refs/tags/v1.15.2.zip # Should change the tag to the latest gtest version

)

# For Windows: Prevent overriding the parent project's compiler/linker settings

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

FetchContent\_MakeAvailable(googletest)

# Configure to build the code

enable\_testing()

# Configure for code-under-test. Build it to a static library

set(SOURCES

../src/hello.cpp

)

add\_library(hello STATIC

  ${SOURCES}

)

# Configure for unittest code. Build it to an executable that uses the static library

add\_executable(hello\_test

  hello\_test.cpp

)

target\_link\_libraries(hello\_test PUBLIC

  GTest::gtest\_main # Built-in main function by gtest

GTest::gmock\_main # Built-in main function by gmock. In this example, we won't use gmock. But in reality, we'll use

hello

)

3. In the test directory, create a file named hello\_test.cpp (as an example) with following content:

#include <gtest/gtest.h>

// In reality, you will need to include hello.cpp here. But let ignore it for simplicity

// A simple testcase without any mock

TEST(HelloTest, BasicAssertions) {

  // Expect two strings not to be equal

  EXPECT\_STRNE("hello", "world");

  // Expect equality

  EXPECT\_EQ(7 \* 6, 42);

}

4. Next create and configure a buildsystem with CMake:

**unittest$ cmake -S . -B build**

-- The C compiler identification is GNU 9.4.0

-- The CXX compiler identification is GNU 9.4.0

-- Check for working C compiler: /usr/bin/cc

-- Check for working C compiler: /usr/bin/cc -- works

-- Detecting C compiler ABI info

-- Detecting C compiler ABI info - done

-- Detecting C compile features

-- Detecting C compile features - done

-- Check for working CXX compiler: /usr/bin/c++

-- Check for working CXX compiler: /usr/bin/c++ -- works

-- Detecting CXX compiler ABI info

-- Detecting CXX compiler ABI info - done

-- Detecting CXX compile features

-- Detecting CXX compile features - done

-- Found Python3: /usr/bin/python3.8 (found version "3.8.10") found components: Interpreter

-- Looking for pthread.h

-- Looking for pthread.h - found

-- Performing Test CMAKE\_HAVE\_LIBC\_PTHREAD

-- Performing Test CMAKE\_HAVE\_LIBC\_PTHREAD - Failed

-- Looking for pthread\_create in pthreads

-- Looking for pthread\_create in pthreads - not found

-- Looking for pthread\_create in pthread

-- Looking for pthread\_create in pthread - found

-- Found Threads: TRUE

-- Configuring done

-- Generating done

-- Build files have been written to: /home/myproject/ut/build

5. Next build your test code

**unittest$ cmake --build build**

Scanning dependencies of target gtest

...

[100%] Built target gtest\_main

6. Finally, run the test executable to see the test result

######### Way 1 #########

**unittest$ cd build && ctest**

Test project /home/myproject/uts/build

Start 1: HelloTest.BasicAssertions

1/1 Test #1: HelloTest.BasicAssertions ........ Passed 0.02 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.07 sec

######### Way 2 (note: the path below is on Windows, it might be different in Linux) #########

**unittest$ build\Debug\hello\_test**

[==========] Running 1 test from 1 test suite.

[----------] Global test environment set-up.

[----------] 1 test from HelloTest

[ RUN ] HelloTest.BasicAssertions

[ OK ] HelloTest.BasicAssertions (0 ms)

[----------] 1 test from HelloTest (1 ms total)

[----------] Global test environment tear-down

[==========] 1 test from 1 test suite ran. (6 ms total)

[ PASSED ] 1 test.

# Basic Concepts

## Assertions

Gtest makes assertions in forms of **macros** that resemble function calls. Tests use assertions to verify the tested code’s behavior.

* If a test crashes or has a failed assertion, then it fails.

When an assertion fails, gtest prints the*source file*, *line number*, and a *failure message*.

* You may also provide a custom failure message by streaming it into the macro using the << operator(s). For example:

ASSERT\_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {

  EXPECT\_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;

}

* If a test succeeds, then it passes.

An assertion’s result can be *success*, *non-fatal failure*, or *fatal failure*.

Below are two types of assertions:

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Note** |
| ASSERT\_\* | Generate **fatal failures** when they fail, and abort the current function. | Because a failed ASSERT\_\* returns immediately, it might skip resource clean-up that comes after it. Thus, cause memory leaks. |
| EXPECT\_\* | Generate **non-fatal failures**, and don’t abort the current function. | This is used in most cases. |

Within these types, there’re various variants of assertions for various testing purposes. For a full list, check [here](http://google.github.io/googletest/reference/assertions.html). For a commonly-used list, check below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Group** | **Type** | **Description** | **Note** |
| Boolean Condition | EXPECT\_TRUE(condition)  ASSERT\_TRUE(condition) | Verifies that *condition* is true. |  |
| EXPECT\_FALSE(condition)  ASSERT\_FALSE(condition) | Verifies that *condition* is false. |  |
| Binary Comparison | EXPECT\_EQ(val1, val2)  ASSERT\_EQ(val1, val2) | Verifies that *val1 == val2*. | * When comparing a pointer to NULL, use EXPECT\_EQ(ptr, nullptr) instead of EXPECT\_EQ(ptr, NULL) * These are used for primitive or object data type. For pointers (e.g. C strings const char\*), check session " C String Comparision " |
| EXPECT\_NE(*val1*, *val2*) ASSERT\_NE(*val1*, *val2*) | Verifies that *val1 != val2*. |  |
| EXPECT\_LT(val1, val2)  ASSERT\_LT(val1, val2) | Verifies that *val1 < val2*. |  |
| EXPECT\_LE(val1, val2)  ASSERT\_LE(val1, val2) | Verifies that *val1 <= val2*. |  |
| EXPECT\_GT(val1, val2)  ASSERT\_GT(val1, val2) | Verifies that *val1 > val2*. |  |
| EXPECT\_GE(val1, val2)  ASSERT\_GE(val1, val2) | Verifies that *val1 >= val2*. |  |
| C String Comparision | EXPECT\_STREQ(*str1*, *str2*) ASSERT\_STREQ(*str1*, *str2*) | Verifies that the two C strings *str1* and *str2* have the same contents. |  |
| EXPECT\_STRNE(str1, str2)  ASSERT\_STRNE(str1, str2) | Verifies that the two C strings str1 and str2 have different contents. |  |
| EXPECT\_STRCASEEQ(str1, str2)  ASSERT\_STRCASEEQ(str1, str2) | Verifies that the two C strings *str1* and *str2* have the same contents, ignoring case. |  |
| EXPECT\_STRCASENE(*str1*, *str2*) ASSERT\_STRCASENE(*str1*, *str2*) | Verifies that the two C strings *str1* and *str2* have different contents, ignoring case |  |
| Floating-Point Comparison | EXPECT\_FLOAT\_EQ(val1, val2)  ASSERT\_FLOAT\_EQ(val1, val2) | Verifies that the two float values *val1* and *val2* are approximately equal, to within 4 ULPs from each other. | Units in the Last Place (ULPs) |
| EXPECT\_DOUBLE\_EQ(*val1*, *val2*) ASSERT\_DOUBLE\_EQ(*val1*, *val2*) | erifies that the two double values *val1* and *val2* are approximately equal, to within 4 ULPs from each other. |  |
| EXPECT\_NEAR(*val1*, *val2*, *abs\_error*) ASSERT\_NEAR(*val1*, *val2*, *abs\_error*) | Verifies that the difference between *val1* and *val2* does not exceed the absolute error bound *abs\_error*. |  |
| Exception Assertions | EXPECT\_THROW(*statement*, *exc\_type*) ASSERT\_THROW(*statement*, *exc\_type*) | Verifies that *statement* throws an exception of type *exception\_type*. | * Code under test can be a compound statement, for example:   EXPECT\_NO\_THROW({  int n = 5;  DoSomething(&n);  }); |
| EXPECT\_ANY\_THROW(*statement*) ASSERT\_ANY\_THROW(*statement*) | Verifies that *statement* throws an exception of any type. |  |
| EXPECT\_NO\_THROW(*statement*) ASSERT\_NO\_THROW(*statement*) | Verifies that *statement* does not throw any exception. |  |
| Predicate Assertions | Advanced assertions for boolean checking, but I think they’re not really necessary in most cases. | | | |
| Generalized Assertion | EXPECT\_THAT(*value*, *matcher*) ASSERT\_THAT(*value*, *matcher*) | Verifies that *value* matches the [matcher](#_Using_Matchers) *matcher* (which is a function object or predicate)  Example:  #include <gmock/gmock.h>  **using** ::testing::AllOf;  **using** ::testing::Gt;  **using** ::testing::Lt;  **using** ::testing::MatchesRegex;  **using** ::testing::StartsWith;  ...  // String "value1" starts with "Hello"?  **EXPECT\_THAT**(value1, StartsWith("Hello"));  // String "value2" matches a regular expression?  **EXPECT\_THAT**(value2, MatchesRegex("Line \\d+"));  // Number "value3" is between 5 and 10  **ASSERT\_THAT**(value3, AllOf(Gt(5), Lt(10)));  // Custom matcher  **using** ::testing::Contains;  **using** ::testing::Property;  **inline** **constexpr** auto HasFoo = [](**const** auto& f) {  return Property("foo", &MyClass::foo, Contains(f));  };  **EXPECT\_THAT**(x, HasFoo("blah")); | * WARNING: Equality matching via EXPECT\_THAT(actual\_value, expected\_value) is supported. However, note that implicit conversions can cause surprising results. For example, EXPECT\_THAT(some\_bool, "some string") will compile and may pass unintentionally. |
| Death Assertions | EXPECT\_DEATH(statement, matcher)  ASSERT\_DEATH(statement, matcher) | Verifies that *statement* causes the process to terminate with a nonzero exit status and produces stderr output that matches *matcher*.  For example, the following code verifies that calling DoSomething(42) causes the process to die with an error message that contains the text My error:  EXPECT\_DEATH(DoSomething(42), "My error"); | The parameter *matcher* is either a matcher for a const std::string&, or a [regular expression](https://google.github.io/googletest/advanced.html#regular-expression-syntax) – a bare string *s* (with no matcher) is treated as [ContainsRegex(s)](https://google.github.io/googletest/reference/matchers.html#string-matchers), not [Eq(s)](https://google.github.io/googletest/reference/matchers.html#generic-comparison).  More about death test in GTest, check [here](https://google.github.io/googletest/advanced.html#death-tests). |
| EXPECT\_EXIT(*statement*, *predicate*, *matcher*) ASSERT\_EXIT(*statement*, *predicate*, *matcher*) | Verifies that *statement* causes the process to terminate with an exit status that satisfies *predicate*, and produces stderr output that matches *matcher*.  The parameter *predicate* is a function or functor that accepts an int exit status and returns a bool. GTest provides 2 predicates to handle common cases:  // Returns true if the program exited normally with the given exit status code  ::testing::ExitedWithCode(exit\_code);  // Returns true if the program was killed by the given signal  // Not available on Windows  ::testing::KilledBySignal(signal\_number);  For example, the following code verifies that calling NormalExit() causes the process to print a message containing the text Success to stderr and exit with exit status code 0:  EXPECT\_EXIT(NormalExit(), ::testing::ExitedWithCode(0), "Success"); |

## Test Suite

## Test Fixture

A test fixture helps reuse the same configuration of objects for different tests. This is extremely useful when writing multiple tests that operate on similar data.

To create a fixture:

1. Derive a class from testing::Test . Start its body with protected:, as we’ll want to access fixture members from sub-classes.
2. Inside the class, declare any objects you plan to use.
3. If necessary, write a default constructor or SetUp() function to prepare the objects for each test.
4. If necessary, write a destructor or TearDown() function to release any resources allocated in SetUp().
5. If needed, define subroutines for your tests to share.

When using a fixture, use TEST**\_F**() instead of TEST() as it allows you to access objects and subroutines in the test fixture. Unlike TEST(), in TEST\_F() the first argument must be the name of the test fixture class. No test suite name is specified for this macro:

TEST\_F(TestFixtureClassName, TestName) {

... test body ...

}

For each test defined with TEST\_F(), GTest will:

1. Create a fresh test fixture at runtime
2. Initialize via SetUp()
3. Run the test
4. Clean up via TearDown()
5. Delete the test fixture

An example of using test fixture is:

#include "this/package/foo.h"

#include <gtest/gtest.h>

// The fixture for testing class Foo.

class FooTest : public testing::Test {

 protected:

    FooTest() {

        // You can do set-up work for each test here.

    }

    ~FooTest() override {

        // You can do clean-up work that doesn't throw exceptions here.

    }

    // If the constructor and destructor are not enough for setting up

    // and cleaning up each test, you can define the following methods:

    void SetUp() override {

        // Code here will be called immediately after the constructor

        // (right before each test).

    }

    void TearDown() override {

        // Code here will be called immediately after each test

        // (right before the destructor).

    }

    // Class members declared here can be used by all tests in the test suite for Foo.

};

// Tests that the Foo::Bar() method does Abc.

TEST\_F(FooTest, MethodBarDoesAbc) {

  const std::string input\_filepath = "this/package/testdata/myinputfile.dat";

  const std::string output\_filepath = "this/package/testdata/myoutputfile.dat";

  Foo f;

  EXPECT\_EQ(f.Bar(input\_filepath, output\_filepath), 0);

}

// Tests that Foo does Xyz.

TEST\_F(FooTest, DoesXyz) {

  // Exercises the Xyz feature of Foo.

}

Briefly, different tests in a test suite have different test fixture objects. **The same test fixture is NOT reused by multiple tests.** Any changes one test makes to the fixture do not affect others.

However, sometimes tests use resources that are expensive to set up, making the one-copy-per-test model prohibitively expensive. If the tests don’t change the resource, there’s no harm in their sharing a single resource copy. So, GTest also supports [per-test-suite set-up/tear-down](https://google.github.io/googletest/advanced.html#sharing-resources-between-tests-in-the-same-test-suite) or [global set-up/tear-down](https://google.github.io/googletest/advanced.html#global-set-up-and-tear-down).

## gMock

Tests should NOT rely on real objects. In other words, dependenciies in tests should be removed as much as possible.

A **mock object implements the same interface as a real object (so it can be used as one), but lets you specify at run time how it will be used and what it should do** (which methods will be called? in which order? how many times? with what arguments? what will they return? etc). As a result, it helps remove unnecessary dependencies in tests and make them fast and reliable.

However, writing mocks manually in C++ is hard. That’s why Gtest includes *gMock* as a library for creating mock classes and using them easily.

For how to write mock objects with gMock, check [this session](#_Mocking).

# Running Tests

## Listing Test Names

Adding flag --gtest\_list\_tests helps list all available tests in a program before running them:

./foo\_test --gtest\_list\_tests

Output:

TestSuite1.

TestName1

TestName2

TestSuite2.

TestName

## Selecting Tests

By default, GTest runs all tests defined. But if you want to run only a subset of the tests, you can set the GTEST\_FILTER environment variable or the --gtest\_filter flag to a filter string. Only tests, whose full names (in the form of TestSuiteName.TestName) match the filter, are run.

For example:

|  |  |
| --- | --- |
| **Concept** | **Description** |
| ./foo\_test | Runs all tests |
| ./foo\_test --gtest\_filter=FooTest.\* | Runs all tests in test suite FooTest |
| ./foo\_test --gtest\_filter=FooTest.Bar | Run only Bar test in test suite FooTest |
| ./foo\_test --gtest\_filter=-\*DeathTest.\* | Runs all non-death tests. |
| ./foo\_test --gtest\_filter=\*Null\*:\*Constructor\* | Runs any test whose full name contains either "Null" or "Constructor" |
| ./foo\_test --gtest\_filter=FooTest.\*-FooTest.Bar | Runs all tests in test suite FooTest except FooTest.Bar |
| ./foo\_test --gtest\_filter=FooTest.\*:BarTest.\*-FooTest.Bar:BarTest.Foo | Runs all tesin test suite FooTest except FooTest.Bar and everything in test suite BarTest except BarTest.Foo |

The format of a filter is a ':'-separated list of wildcard patterns (called the positive patterns) optionally followed by a '-' and another ':'-separated pattern list (called the negative patterns). A test matches the filter if and only if it matches any of the positive patterns but does not match any of the negative patterns.

A pattern may contain '\*' (matches any string) or '?' (matches any single character)

## Colored Terminal Output

You can set the GTEST\_COLOR environment variable or the --gtest\_color command line flag to yes, no, or auto (the default) to **enable colors**, disable colors, or let GTest decide.

...

[----------] 1 test from FooTest

[ RUN ] FooTest.DoesAbc

[ OK ] FooTest.DoesAbc

[----------] 2 tests from BarTest

[ RUN ] BarTest.HasXyzProperty

[ OK ] BarTest.HasXyzProperty

[ RUN ] BarTest.ReturnsTrueOnSuccess

... some error messages ...

[ FAILED ] BarTest.ReturnsTrueOnSuccess

...

[==========] 30 tests from 14 test suites ran.

[ PASSED ] 28 tests.

[ FAILED ] 2 tests, listed below:

[ FAILED ] BarTest.ReturnsTrueOnSuccess

[ FAILED ] AnotherTest.DoesXyz

2 FAILED TESTS

## Suppressing Test Passes

By default, GTest prints 1 line of output for each test, indicating if it passed or failed. To **show only test failures**, run the test program with --gtest\_brief=1, or set the GTEST\_BRIEF environment variable to 1.

## Repeating the Tests

<https://google.github.io/googletest/advanced.html#repeating-the-tests>

## Shuffling the Tests

<https://google.github.io/googletest/advanced.html#shuffling-the-tests>

## Generating XML/JSON Reports

<https://google.github.io/googletest/advanced.html#generating-an-xml-report>

## Controlling How Failures Are Reported

<https://google.github.io/googletest/advanced.html#controlling-how-failures-are-reported>

# Writing Tests

## Invoking Tests

TEST() and TEST\_F() are implicitly registered. So, **you don’t have to re-list all your defined tests in order to run them**.

After defining your tests, you can run them with RUN\_ALL\_TESTS(), which returns 0 if all the tests are successful, or 1 otherwise. Note that RUN\_ALL\_TESTS() runs all tests in your link unit–they can be from different test suites, or even different source files.

## Writing the main() Function

Most users should NOT need to write their own main function, and instead, link with gtest\_main (as opposed to with gtest), which defines a suitable entry point.

<https://google.github.io/googletest/primer.html#writing-the-main-function>

## Print Values

<https://google.github.io/googletest/advanced.html#teaching-googletest-how-to-print-your-values>

## Using Assertions in Sub-Routines

If a certain logic is repeated in multiple tests, it can be put into a subroutine which will be then called by tests.

But note that when ASSERT\_\* fails, it only abort the current function (in this case, the sub-routine), not the entire test.

Example 1:

void Subroutine() {

  // Generates a fatal failure and aborts the current function.

  ASSERT\_EQ(1, 2);

  // The following won't be executed.

  ...

}

TEST(FooTest, Bar) {

  Subroutine();

  // The following will be executed even when ASSERT\_EQ returns false

  ...

}

Example 2: If you want to abort the entire test when ASSERT\_\* in sub-routines fails

testing::Test::HasFatalFailure() in the class returns true if an assertion in the current test has suffered a fatal failure. This allows functions to catch fatal failures in a sub-routine and return early.

class Test {

 public:

  ...

  static bool HasFatalFailure();

};

TEST(FooTest, Bar) {

  Subroutine();

  // Aborts if Subroutine() had a fatal failure.

  if (HasFatalFailure()) return;

  // The following won't be executed.

  ...

}

If HasFatalFailure() is used outside of TEST(), TEST\_F(), or a test fixture, you must add the ::testing::Test:: prefix, as in:

if (testing::Test::HasFatalFailure()) return;

## Typed Tests

Typed tests are useful when you want to **test the same logic for different types** (e.g. C++ templates). Otherwise, you have to write m\*n TEST or TEST\_F to test m tests over n types. That’s horrible!

Here’s how you do it:

1. Define a fixture class template:

template <typename T>

class FooTest : public testing::Test {

 public:

  ...

  using List = std::list<T>;

  static T shared\_;

  T value\_;

};

1. Associate a list of types with the test suite, which will be repeated for each type in the list:

using MyTypes = ::testing::Types<char, int, unsigned int>;

TYPED\_TEST\_SUITE(FooTest, MyTypes);

The type alias (using or typedef) is necessary for the TYPED\_TEST\_SUITE macro to parse correctly. Otherwise, the compiler will think that each comma in the type list introduces a new macro argument.

1. Use TYPED\_TEST() instead of TEST\_F() to define a typed test for this test suite. You can repeat this as many times as you want:

TYPED\_TEST(FooTest, DoesBlah) {

  // Inside a test, refer to the special name TypeParam to get the type

  // parameter.  Since we are inside a derived class template, C++ requires

  // us to visit the members of FooTest via 'this'.

  TypeParam n = this->value\_;

  // To visit static members of the fixture, add the 'TestFixture::'

  // prefix.

  n += TestFixture::shared\_;

  // To refer to typedefs in the fixture, add the 'typename TestFixture::'

  // prefix.  The 'typename' is required to satisfy the compiler.

  typename TestFixture::List values;

  values.push\_back(n);

  ...

}

TYPED\_TEST(FooTest, HasPropertyA) { ... }

You can see [sample6\_unittest.cc](https://github.com/google/googletest/blob/main/googletest/samples/sample6_unittest.cc) for a complete example.

## Handling Test Events

GTest provides an event listener API to let you **receive notifications about the progress of a test program and test failures**. The events you can listen to include the start and end of the test program, a test suite, or a test method, among others.

<https://google.github.io/googletest/advanced.html#extending-googletest-by-handling-test-events>

## Value-Parameterized Tests

Value-parameterized tests allow you to test your code with different parameters without writing multiple copies of the same test.

<https://google.github.io/googletest/advanced.html#value-parameterized-tests>

## Using Matchers

<https://google.github.io/googletest/reference/matchers.html>

## Using Actions

Actions specify what a mock function should do when invoked. This page lists the built-in actions provided by GoogleTest. All actions are defined in the ::testing namespace.

<https://google.github.io/googletest/reference/actions.html>

## Mocking with gMock

### Quick Example

1. Suppose your Turtle class has following methods:

class Turtle {

  ...

  virtual ~Turtle() {}

  virtual void PenUp() = 0;

  virtual void PenDown() = 0;

  virtual void Forward(int distance) = 0;

  virtual void Turn(int degrees) = 0;

  virtual void GoTo(int x, int y) = 0;

  virtual int GetX() const = 0;

  virtual int GetY() const = 0;

};

2. You can create a mock class for Turtle with following code:

#include <gmock/gmock.h>

class MockTurtle : public Turtle {

 public:

  ...

  MOCK\_METHOD(void, PenUp, (), (override));

  MOCK\_METHOD(void, PenDown, (), (override));

  MOCK\_METHOD(void, Forward, (int distance), (override));

  MOCK\_METHOD(void, Turn, (int degrees), (override));

  MOCK\_METHOD(void, GoTo, (int x, int y), (override));

  MOCK\_METHOD(int, GetX, (), (const, override));

  MOCK\_METHOD(int, GetY, (), (const, override));

};

You don’t need to define these mock methods because the MOCK\_METHOD macro will generate the definitions for you. It’s that simple.

3. Now you have a mock class, you need to use it.

Here’s an example:

#include "path/to/mock-turtle.h"

#include <gmock/gmock.h>

#include <gtest/gtest.h>

using ::testing::AtLeast;

TEST(PainterTest, CanDrawSomething) {

  MockTurtle turtle;                              // Create a mock object for Turtle

**EXPECT\_CALL**(turtle, PenDown())                  // Tell gMock that "I expect PenDown() will be called at lease one time"

      .Times(AtLeast(1));

  Painter painter(&turtle);

**EXPECT\_TRUE**(painter.DrawCircle(0, 0, 10));      // Tell gTest that "I expect DrawCircle(...) will return true"

}

This test checks that PenDown() is called at least once. If the painter object didn’t call this method, your test will fail with a message like this:

path/to/my\_test.cc:119: Failure

Actual function call count doesn't match this expectation:

Actually: never called;

Expected: called at least once.

Stack trace:

...

**Notes**:

* In gMock we use the EXPECT\_CALL() macro to set an expectation on a mock method.
* **When a mock is destructed, gMock will automatically check whether all expectations on it have been satisfied**. So, if your mock objects are never deleted, the final verification won’t happen. In case you want to verify the expectations on a mock object earlier, check [here](https://google.github.io/googletest/gmock_cheat_sheet.html#verifying-and-resetting-a-mock).
* gMock requires expectations to be set before the mock functions are called, otherwise the behavior is undefined. This means EXPECT\_CALL() should be read as expecting that a call will occur in the future, not that a call has occurred.

### Creating Mock Classes

#### General Syntax

Mock classes are defined as normal classes, using the MOCK\_METHOD macro to generate mocked methods. The macro gets 3 or 4 parameters:

class MyMock {

public:

MOCK\_METHOD(ReturnType, MethodName, (Args...));

MOCK\_METHOD(ReturnType, MethodName, (Args...), (Specs...));

};

* The first 3 parameters are simply the method declaration, split into 3 parts.
* The 4th parameter accepts a comma-separated list of qualifiers, which affect the generated method:
  + const - Makes the mocked method a const method. Required if overriding a const method.
  + override - Marks the method with override. Recommended if overriding a virtual method.
  + noexcept - Marks the method with noexcept. Required if overriding a noexcept method.
  + Calltype(...) - Sets the call type for the method (e.g. to STDMETHODCALLTYPE), useful in Windows.
  + ref(...) - Marks the method with the reference qualification specified. Required if overriding a method that has reference qualifications. Eg ref(&) or ref(&&)

**NOTES**:

* All gMock symbols are in the testing namespace unless they are macros.
* **MOCK\_METHOD must be public**, regardless of the method being mocked being public, protected, or private in the base class. This allows ON\_CALL and EXPECT\_CALL to call the mock function from outside of the mock class.

class Foo {

 public:

  virtual bool Transform(Gadget\* g) = 0;

 protected:

  virtual void Resume();

 private:

  int GetTimeOut(int time);

  const Bar& GetBar() const;

};

class MockFoo : public Foo {

 public:

  MOCK\_METHOD(bool, Transform, (Gadget\* g), (override));

  MOCK\_METHOD(void, Resume, (), (override));

  MOCK\_METHOD(int, GetTimeOut, (int time), ());

  MOCK\_METHOD(const Bar&, GetBar, (), (const));

};

* For virtual methods, the suggestion is to add the override keyword to the 4th parameter of MOCK\_METHOD.
* The syntax for mocking non-virtual methods is the same as mocking virtual methods (just don’t add override)
* For const methods, add const keyword to the 4th parameter of MOCK\_METHOD..
* You don’t need to mock all methods in the real class if your test doesn’t call all methods.
* Generally, you **should NOT mock classes you don’t own**. Applying *Dependency Injection* design principle is important. That’s said you can introduce a thin layer FooAdaptor on top of Foo and code to this new interface. Since you own FooAdaptor, you can absorb changes in Foo much more easily. And the nice thing is that you can mock FooAdaptor.
* How to deal with unprotected commas? Unprotected commas, i.e. commas in templates, prevent MOCK\_METHOD from parsing its arguments correctly:

class MockFoo {

public:

MOCK\_METHOD(std::pair<bool, int>, GetPair, ()); // Won't compile!

MOCK\_METHOD(bool, CheckMap, (std::map<int, double>, bool)); // Won't compile!

};

Solution 1: Wrap with parentheses:

class MockFoo {

public:

MOCK\_METHOD((std::pair<bool, int>), GetPair, ());

MOCK\_METHOD(bool, CheckMap, ((std::map<int, double>), bool));

};

Solution 2: Define an alias:

class MockFoo {

public:

using BoolAndInt = std::pair<bool, int>;

MOCK\_METHOD(BoolAndInt, GetPair, ());

using MapIntDouble = std::map<int, double>;

MOCK\_METHOD(bool, CheckMap, (MapIntDouble, bool));

};

**IMPOTANTS**:

Before the generic MOCK\_METHOD macro above, there were a old version of it, called MOCK\_METHODn. These macros are still supported, though migration to the new one is recommended. For more details, check [here](https://google.github.io/googletest/gmock_cook_book.html#old-style-mock_methodn-macros).

#### Mocking Ctor and Dtor

Mocking **constructor and destructor is NOT supported**.

#### Mocking Class Templates

You can mock class templates just like any class.

template <typename Elem>

class StackInterface {

  // Must be virtual as we'll inherit from StackInterface.

  virtual ~StackInterface();

  virtual int GetSize() const = 0;

  virtual void Push(const Elem& x) = 0;

};

template <typename Elem>

class MockStackInterface : public StackInterface<Elem> {

  MOCK\_METHOD(int, GetSize, (), (override));

  MOCK\_METHOD(void, Push, (const Elem& x), (override));

};

#### Mocking Non-Virtual Functions

<https://google.github.io/googletest/gmock_cook_book.html#MockingNonVirtualMethods>

<https://google.github.io/googletest/gmock_cook_book.html#alternative-to-mocking-concrete-classes>

<https://www.sandordargo.com/blog/2022/03/09/mocking-non-virtual-and-free-functions>

<https://github.com/hedayat/powerfake> (third-party)

#### Mocking Free Functions (i.e. a C-style functions or a static methods)

It is IMPOSIBLE to directly mock a free function.

**Solution 1**: Rewrite your code to use an interface or abstract class.

Instead of calling a free function (say, OpenFile) directly, introduce an interface (say, FileInterface) for it and have a concrete subclass that calls the free function. Your code should talk to FileInterface to open a file:

class FileInterface {

 public:

  virtual bool Open(const char\* path, const char\* mode) = 0;

};

class File : public FileInterface {

 public:

  bool Open(const char\* path, const char\* mode) override {

     return OpenFile(path, mode);

  }

};

Now it’s easy to mock out the function.

**Solution 2**: Rewrite your code to accept a std::function instead of the free function, and then use MockFunction to mock the std::function.

#### Dealing with Uninteresting Call

Mock methods, having no EXPECT\_CALL spec but called, are named as "uninteresting call". This, by default, cause gMock to print a warning.

But if you want to ignore uninteresting call, check [here](https://google.github.io/googletest/gmock_cook_book.html#NiceStrictNaggy).

### Setting Expectations

The key to using a mock object successfully is to set the right expectations on it.

#### General Syntax

The general syntax of EXPECT\_CALL() is:

**EXPECT\_CALL**(MockObjectName, MethodName (*matchers*))

.**With**(multi\_argument\_matcher) // Can be used at most once

.**Times**(cardinality) // Can be used at most once

.**InSequence**(sequences...) // Can be used any number of times

.**After**(expectations...) // Can be used any number of times

.**WillOnce**(action) // Can be used any number of times

.**WillRepeatedly**(action) // Can be used at most once

.**RetiresOnSaturation**(); // Can be used at most once

The macro has 2 arguments:

* The mock object MockOBjectName
* The method name MethodName following its arguments which match a given matchers.
  + Matcher is a comma-separated list of [matchers](#_Using_Matchers) that corresponding to each argument of the method. The expectation will apply only to calls of MethodName whose arguments match all of the matchers.
  + If (matchers...) is omitted, the expectation behaves as if each argument’s matcher were a wildcard matcher (\_). This syntax allows you to specify "called with any arguments" without explicitly specifying the number or types of arguments. Very convenient!
  + **Brieftly, in case of overloaded methods, matchers must be specified. Otherwise, you can omit it if you don’t need it**.
  + All built-in matchers are defined in the ::testing namespace.

In addition, there are *modifier clauses* following EXPECT\_CALL(). They are used to modify the expectation.

|  |  |  |
| --- | --- | --- |
| **Clause** | **Description** | **Note** |
| .**With**(multi\_argument\_matcher) | <https://google.github.io/googletest/reference/mocking.html#EXPECT_CALL.With> |  |
| .**Times**(cardinality) | Specifies **how many times the mock function call is expected**.   |  |  | | --- | --- | | **Cardinality** | **Meaning** | | AnyNumber() | Called any number of times. | | AtLeast(n) | Called at least *n* times. | | AtMost(n) | Called at most *n* times. | | Between(m, n) | Called between *m* and *n* times, inclusive. | | Exactly(n) or n | Called exactly *n* times. If *n* is 0, the call should never happen. | | * Mostly used for functions with no return value. |
| .**InSequence**(sequences...) | <https://google.github.io/googletest/reference/mocking.html#EXPECT_CALL.InSequence> |  |
| .**After**(expectations...) | <https://google.github.io/googletest/reference/mocking.html#EXPECT_CALL.After> |  |
| .**WillOnce**(action) | Specifies the **mock function’s actual behavior when invoked**, for a **single** matching function call.  The parameter action represents the [action](#_Using_Actions) that the function call will perform. | * Mostly used for functions with return value. |
| .**WillRepeatedly**(action) | Specifies the **mock function’s actual behavior when invoked**, for all **subsequent** matching function calls.  The parameter action represents the action that the function call will perform. | * Mostly used for functions with return value. |
| .**RetiresOnSaturation**() | <https://google.github.io/googletest/reference/mocking.html#EXPECT_CALL.RetiresOnSaturation> |  |

#### Cardinalities: How Many Times Will It Be Called?

Methods, like Times(), WillOnce(), WillRepeatedly(), etc., are called a cardinality as it tells how many times the call should occur. It allows us to repeat an expectation many times without actually writing it as many times.

Each matching function call will perform the next action in the order declared.

Example 1: The following code says that the Turtle object’s GetX() method will be called five times, it will return 100 the first time, 150 the second time, and then 200 every time:

using ::testing::Return;

...

TEST(TurtleWrapper, GetXYZ) {

Turtle turtle;

**EXPECT\_CALL**(turtle, GetX())

.Times(5)

.WillOnce(Return(100))

.WillOnce(Return(150))

.WillRepeatedly(Return(200));

}

Example 2: The following code says that Turtle.GetX() is expected to be called exactly 3 times and will return 100, 200, and 300 respectively:

using ::testing::Return;

...

TEST(TurtleWrapper, GetXYZ) {

Turtle turtle;

**EXPECT\_CALL**(turtle, GetX())

.WillOnce(Return(100))

.WillOnce(Return(200))

.WillOnce(Return(300));

}

Example 3: If you think the two testcases GetXYZ\_TC1 and GetXYZ\_TC2 below expect Turtle.GetX() is to be called exactly 1 time for each, then you're wrong. Actually, Turtle.GetX() will be called twice in GetXYZ\_TC2. The reason is that turtle is a global variable.

using ::testing::Return;

...

Turtle turtle; // global variable

TEST(TurtleWrapper, GetXYZ\_TC1) {

**EXPECT\_CALL**(turtle, GetX())

.Times(1);

}

TEST(TurtleWrapper, GetXYZ\_TC2) {

**EXPECT\_CALL**(turtle, GetX())

.Times(1); // actual call time is 2, not 1

}

#### Matchers: What Arguments Do We Expect?

When a mock function takes arguments, we may specify what arguments we are expecting.

Example 4:

In Example 1, 100, 150 and 200 are matchers. Implicitly, they are the same as Eq(100), Eq(150) and Eq(200), which specify that the argument must be equal (using operator==) to the matcher argument. There are many built-in matchers for common types (as well as custom matchers). For example:

using ::testing::Ge;

...

// Expects that Forward must be called with value at least 100.

**EXPECT\_CALL**(turtle, Forward(Ge(100)));

Example 5:

If you aren’t interested in the value of an argument, write \_ as the argument or write nothing, which means "anything goes".

// Way 1: Expects that Forward must be called with value 100.

**EXPECT\_CALL**(turtle, Forward(100));

// Way 2: Expects that Forward must be called with any value.

**EXPECT\_CALL**(turtle, Forward(\_));

// Way 3: Expects that Forward must be called with any value.

**EXPECT\_CALL**(turtle, Forward);

##### Dealing with Overloaded Functions

**Case 1**: Overloaded functions with the same number of arguments

You may need to specify the exact type of a matcher, either by wrapping your matcher in testing::Matcher<type>(), or using a matcher whose type is fixed (testing::TypedEq<type>, testing::An<type>(), etc):

using ::testing::An;

using ::testing::Matcher;

using ::testing::TypedEq;

class MockPrinter : public Printer {

 public:

  MOCK\_METHOD(void, Print, (int n), (override));

  MOCK\_METHOD(void, Print, (char c), (override));

};

TEST(PrinterTest, Print) {

  MockPrinter printer;

**EXPECT\_CALL**(printer, Print(An<int>()));            // void Print(int);

**EXPECT\_CALL**(printer, Print(Matcher<int>(Lt(5))));  // void Print(int);

**EXPECT\_CALL**(printer, Print(TypedEq<char>('a')));   // void Print(char);

  printer.Print(3);

  printer.Print(6);

  printer.Print('a');

}

**Case 2**: Functions overloaded on the const-ness of this object

You can use the Const() argument wrapper.

using ::testing::ReturnRef;

class MockFoo : public Foo {

  MOCK\_METHOD(Bar&, GetBar, (), (override));

  MOCK\_METHOD(const Bar&, GetBar, (), (const, override));

};

...

  MockFoo foo;

  Bar bar1, bar2;

**EXPECT\_CALL**(foo, GetBar())         // The non-const GetBar().

      .WillOnce(ReturnRef(bar1));

**EXPECT\_CALL**(Const(foo), GetBar())  // The const GetBar().

      .WillOnce(ReturnRef(bar2));

#### Actions: What Should It Do?

In above examples, you already saw the action passed to WillOnce(...) or WillRepeatedly(...). Generally, for functions with return value, the action is returning a value (details are shown in the table below); and for functions without return value, the action is nothing.

|  |  |  |
| --- | --- | --- |
| **Action** | **Description** | **Note** |
| Return() | Return from a void mock function. |  |
| Return(value) | Return value.  If the type of value is different to the mock function’s return type, value is converted to the latter type *at the time the expectation is set*, not when the action is executed. |  |
| ReturnArg<N>() | Return the N-th (0-based) argument. | * Mostly used for function’s output parameter |
| ReturnNew<T>(a1, ..., ak) | Return new T(a1, ..., ak); a different object is created each time. |  |
| ReturnNull() | Return a null pointer. |  |
| ReturnPointee(ptr) | Return the value pointed to by ptr. |  |
| ReturnRef(variable) | Return a reference to variable. |  |
| ReturnRefOfCopy(value) | Return a reference to a copy of value; the copy lives as long as the action. |  |
| ReturnRoundRobin({a1, ..., ak}) | Each call will return the next ai in the list, starting at the beginning when the end of the list is reached. |  |

##### Returning with Side Effects

<https://google.github.io/googletest/reference/actions.html#side-effects>

##### Describe, But Don’t Assert with ON\_CALL

In addition to returning values, there are cases you'll need macro ON\_CALL(MockObjectName, MethodName(matchers...)). ON\_CALL is that like EXPECT\_CALL, except that it **does not set any expectations for the method** to be called (so it doesn’t make sure that the method gets called).

In addition, there are *modifier clauses* following ON\_CALL():

|  |  |  |
| --- | --- | --- |
| **Clause** | **Description** | **Note** |
| .With(multi\_argument\_matcher) | Restricts the specification to only mock function calls whose arguments as a whole match the multi-argument matcher multi\_argument\_matcher. | This clause **must be used exactly once** with each ON\_CALL statement. Otherwise, runtime error. |
| .WillByDefault(action) | Specifies the default behavior of a matching mock function call.  The action specified by this clause is replaced by the actions specified on a matching EXPECT\_CALL statement. | This clause **must be used exactly once** with each ON\_CALL statement. Otherwise, runtime error. |

Example 1:

The following code sets the default behavior when Turtel.SetPositon() is called with any two arguments, the first argument being less than the second:

using ::testing::\_;

using ::testing::Lt;

using ::testing::Return;

...

ON\_CALL(Turtel, SetPosition(\_, \_))

    .With(Lt())

    .WillByDefault(Return(true));

Example 2:

The following code specifies that by default, a call to Greedy.Greet() will return "hello":

using ::testing::Return;

...

ON\_CALL(Greedy, Greet())

    .WillByDefault(Return("hello"));

**QA**:

**ON\_CALL and EXPECT\_CALL: What to use?**

As mentioned, the biggest difference between ON\_CALL and EXPECT\_CALL is that ON\_CALL doesn’t set any expectations.

It might sound counter-intuitive, but because of the above difference, you should use ON\_CALL by default.

With EXPECT\_CALL you might overspecify your tests and they become too brittle. You might couple the tests too closely to the implementation. Think about the problem of test contra-variance explained by Uncle Bob.

Use EXPECT\_CALL only when the main purpose of a test is to make sure that something gets called, and even then you should think twice whether you want it to be tested at all.

##### Using a Function, Functor, or Lambda as an Action

<https://google.github.io/googletest/gmock_cook_book.html#FunctionsAsActions>

<https://google.github.io/googletest/reference/actions.html#using-a-function-functor-or-lambda-as-an-action>

##### Performing Different Actions Based on the Arguments

You can make a method do different things depending on its argument values:

using ::testing::\_;

using ::testing::Lt;

using ::testing::Return;

...

  // The default case.

  EXPECT\_CALL(foo, DoThis(\_))

      .WillRepeatedly(Return('b'));

  // The more specific case.

  EXPECT\_CALL(foo, DoThis(Lt(5)))

      .WillRepeatedly(Return('a'));

Now, if foo.DoThis() is called with a value less than 5, 'a' will be returned; otherwise 'b' will be returned.

##### Matching Multiple Arguments as a Whole

<https://google.github.io/googletest/gmock_cook_book.html#matching-multiple-arguments-as-a-whole>

#### Using Multiple Expectations

**gMock will search the expectations in the reverse order they are defined**, and stop when an active expectation that matches the arguments is found (you can think of it as "newer rules override older ones."). If the matching expectation cannot take any more calls, you will get an upper-bound-violated failure. Here’s an example:

using ::testing::\_;

...

EXPECT\_CALL(turtle, Forward(\_));            // #1

EXPECT\_CALL(turtle, Forward(10)).Times(2);  // #2

If Forward(10) is called three times in a row, the third time it will be an error, as the last matching expectation (#2) has been saturated. If, however, the third Forward(10) call is replaced by Forward(20), then it would be OK, as now #1 will be the matching expectation.

### Setting Default Actions

<https://github.com/google/googletest/blob/main/docs/gmock_cheat_sheet.md#setting-default-actions-oncall>

### Running gMock in Tests

|  |  |
| --- | --- |
| **Flag** | **Description** |
| --gmock\_catch\_leaked\_mocks=0 | Don’t report leaked mock objects as failures. |
| --gmock\_verbose=LEVEL | Sets the default verbosity level (info, warning, or error) of gMock messages |

# Gcovr

<https://gcovr.com/en/stable/guide.html>

Gcovr = [gcov](http://gcc.gnu.org/onlinedocs/gcc/Gcov.html) + summarized code coverage results in various formats (HTML, XML – compatible with Jenkins, JSON, etc.).

# Lcov