Googletest Primer

Introduction: Why googletest?

googletest helps you write better C++ tests.

googletest is a testing framework developed by the Testing Technology team with Google's specific requirements and constraints in mind. Whether you work on Linux, Windows, or a Mac, if you write C++ code, googletest can help you. And it supports *any* kind of tests, not just unit tests.

So what makes a good test, and how does googletest fit in? We believe:

1. Tests should be *independent* and *repeatable*. It's a pain to debug a test that succeeds or fails as a result of other tests. googletest isolates the tests by running each of them on a different object. When a test fails, googletest allows you to run it in isolation for quick debugging.
2. Tests should be well *organized* and reflect the structure of the tested code. googletest groups related tests into test suites that can share data and subroutines. This common pattern is easy to recognize and makes tests easy to maintain. Such consistency is especially helpful when people switch projects and start to work on a new code base.
3. Tests should be *portable* and *reusable*. Google has a lot of code that is platform-neutral; its tests should also be platform-neutral. googletest works on different OSes, with different compilers, with or without exceptions, so googletest tests can work with a variety of configurations.
4. When tests fail, they should provide as much *information* about the problem as possible. googletest doesn't stop at the first test failure. Instead, it only stops the current test and continues with the next. You can also set up tests that report non-fatal failures after which the current test continues. Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.
5. The testing framework should liberate test writers from housekeeping chores and let them focus on the test *content*. googletest automatically keeps track of all tests defined, and doesn't require the user to enumerate them in order to run them.
6. Tests should be *fast*. With googletest, you can reuse shared resources across tests and pay for the set-up/tear-down only once, without making tests depend on each other.

Since googletest is based on the popular xUnit architecture, you'll feel right at home if you've used JUnit or PyUnit before. If not, it will take you about 10 minutes to learn the basics and get started. So let's go!

Beware of the nomenclature

{: .callout .note}
Note: There might be some confusion arising from different definitions of the terms *Test*, *Test Case* and *Test Suite*, so beware of misunderstanding these.

Historically, googletest started to use the term *Test Case* for grouping related tests, whereas current publications, including International Software Testing Qualifications Board (ISTQB) materials and various textbooks on software quality, use the term *Test Suite* for this.

The related term *Test*, as it is used in googletest, corresponds to the term *Test Case* of ISTQB and others.

The term *Test* is commonly of broad enough sense, including ISTQB's definition of *Test Case*, so it's not much of a problem here. But the term *Test Case* as was used in Google Test is of contradictory sense and thus confusing.

googletest recently started replacing the term *Test Case* with *Test Suite*. The preferred API is *TestSuite*. The older TestCase API is being slowly deprecated and refactored away.

So please be aware of the different definitions of the terms:

Meaning	googletest Term	ISTQB Term
Exercise a particular program path with specific input values and verify the results	TEST()	Test Case

Basic Concepts

When using googletest, you start by writing *assertions*, which are statements that check whether a condition is true. An assertion's result can be *success*, *nonfatal failure*, or *fatal failure*. If a fatal failure occurs, it aborts the current function; otherwise the program continues normally.

Tests use assertions to verify the tested code's behavior. If a test crashes or has a failed assertion, then it *fails*; otherwise it *succeeds*.

A *test suite* contains one or many tests. You should group your tests into test suites that reflect the structure of the tested code. When multiple tests in a test suite need to share common objects and subroutines, you can put them into a *test fixture* class.

A *test program* can contain multiple test suites.

We'll now explain how to write a test program, starting at the individual assertion level and building up to tests and test suites.

Assertions

googletest assertions are macros that resemble function calls. You test a class or function by making assertions about its behavior. When an assertion fails, googletest prints the assertion's source file and line number location, along with a failure message. You may also supply a custom failure message which will be appended to googletest's message.

The assertions come in pairs that test the same thing but have different effects on the current function. `ASSERT_*` versions generate fatal failures when they fail, and **abort the current function**. `EXPECT_*` versions generate nonfatal failures, which don't abort the current function. Usually `EXPECT_*` are preferred, as they allow more than one failure to be reported in a test. However, you should use `ASSERT_*` if it doesn't make sense to continue when the assertion in question fails.

Since a failed `ASSERT_*` returns from the current function immediately, possibly skipping clean-up code that comes after it, it may cause a space leak. Depending on the nature of the leak, it may or may not be worth fixing - so keep this in mind if you get a heap checker error in addition to assertion errors.

To provide a custom failure message, simply stream it into the macro using the `<<` operator or a sequence of such operators. See the following example, using the `ASSERT_EQ` and `EXPECT_EQ` macros to verify value equality:

```
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;
}
```

Anything that can be streamed to an `ostream` can be streamed to an assertion macro--in particular, C strings and `string` objects. If a wide string (`wchar_t*`, `TCHAR*` in `UNICODE` mode on Windows, or `std::wstring`) is streamed to an assertion, it will be translated to UTF-8 when printed.

GoogleTest provides a collection of assertions for verifying the behavior of your code in various ways. You can check Boolean conditions, compare values based on relational operators, verify string values, floating-point values, and much more. There are even assertions that enable you to verify more complex states by providing custom predicates. For the complete list of assertions provided by GoogleTest, see the Assertions Reference.

Simple Tests

To create a test:

1. Use the `TEST()` macro to define and name a test function. These are ordinary C++ functions that don't return a value.
2. In this function, along with any valid C++ statements you want to include, use the various googletest assertions to check values.
3. The test's result is determined by the assertions; if any assertion in the test fails (either fatally or non-fatally), or if the test crashes, the entire test fails. Otherwise, it succeeds.

```
TEST(TestSuiteName, TestName) {
  ... test body ...
}
```

`TEST()` arguments go from general to specific. The *first* argument is the name of the test suite, and the *second* argument is the test's name within the test suite. Both names must be valid C++ identifiers, and they should not contain any underscores (`_`). A test's *full name* consists of its containing test suite and its individual name. Tests from different test suites can have the same individual name.

For example, let's take a simple integer function:

```
int Factorial(int n);  // Returns the factorial of n
```

A test suite for this function might look like:

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
  EXPECT_EQ(Factorial(0), 1);
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(Factorial(1), 1);
  EXPECT_EQ(Factorial(2), 2);
  EXPECT_EQ(Factorial(3), 6);
  EXPECT_EQ(Factorial(8), 40320);
}
```

googletest groups the test results by test suites, so logically related tests should be in the same test suite; in other words, the first argument to their `TEST()` should be the same. In the above example, we have two tests, `HandlesZeroInput` and `HandlesPositiveInput`, that belong to the same test suite `FactorialTest`.

When naming your test suites and tests, you should follow the same convention as for
naming functions and classes.

Availability: Linux, Windows, Mac.

Test Fixtures: Using the Same Data Configuration for Multiple Tests {#same-data-multiple-tests}

If you find yourself writing two or more tests that operate on similar data, you can use a *test fixture*. This allows you to reuse the same configuration of objects for several different tests.

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. A common mistake is to spell `SetUp()` as

`Setup()` with a small `u` - Use `override` in C++11 to make sure you spelled it correctly.
4. If necessary, write a destructor or `TearDown()` function to release any resources you allocated in `SetUp()` . To learn when you should use the constructor/destructor and when you should use `SetUp()/TearDown()`, read the FAQ.
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:

```
TEST_F(TestFixtureName, TestName) {
  ... test body ...
}
```

Like `TEST()`, the first argument is the test suite name, but for `TEST_F()` this must be the name of the test fixture class. You've probably guessed: `_F` is for fixture.

Unfortunately, the C++ macro system does not allow us to create a single macro that can handle both types of tests. Using the wrong macro causes a compiler error.

Also, you must first define a test fixture class before using it in a `TEST_F()`, or you'll get the compiler error "`virtual outside class declaration`".

For each test defined with `TEST_F()`, googletest will create a *fresh* test fixture at runtime, immediately initialize it via `SetUp()`, run the test, clean up by calling `TearDown()`, and then delete the test fixture. Note that different tests in the same test suite have different test fixture objects, and googletest always deletes a test fixture before it creates the next one. googletest does **not** reuse the same test fixture for multiple tests. Any changes one test makes to the fixture do not affect other tests.

As an example, let's write tests for a FIFO queue class named `Queue`, which has the following interface:

```
template <typename E>  // E is the element type.
class Queue {
 public:
  Queue();
  void Enqueue(const E& element);
  E* Dequeue();  // Returns NULL if the queue is empty.
  size_t size() const;
  ...
};
```

First, define a fixture class. By convention, you should give it the name `FooTest` where `Foo` is the class being tested.

```
class QueueTest : public ::testing::Test {
 protected:
```

```
  void SetUp() override {
     q1_.Enqueue(1);
     q2_.Enqueue(2);
     q2_.Enqueue(3);
  }

  // void TearDown() override {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

In this case, `TearDown()` is not needed since we don't have to clean up after each test, other than what's already done by the destructor.

Now we'll write tests using `TEST_F()` and this fixture.

```
TEST_F(QueueTest, IsEmptyInitially) {
  EXPECT_EQ(q0_.size(), 0);
}

TEST_F(QueueTest, DequeueWorks) {
  int* n = q0_.Dequeue();
  EXPECT_EQ(n, nullptr);

  n = q1_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 1);
  EXPECT_EQ(q1_.size(), 0);
  delete n;

  n = q2_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 2);
  EXPECT_EQ(q2_.size(), 1);
  delete n;
}
```

The above uses both `ASSERT_*` and `EXPECT_*` assertions. The rule of thumb is to use `EXPECT_*` when you want the test to continue to reveal more errors after the assertion failure, and use `ASSERT_*` when continuing after failure doesn't make sense. For example, the second assertion in the `Dequeue` test is `ASSERT_NE(n, nullptr)`, as we need to dereference the pointer `n` later, which would lead to a segfault when `n` is `NULL`.

When these tests run, the following happens:

1. googletest constructs a `QueueTest` object (let's call it `t1`).
2. `t1.SetUp()` initializes `t1`.
3. The first test (`IsEmptyInitially`) runs on `t1`.
4. `t1.TearDown()` cleans up after the test finishes.
5. `t1` is destructed.

6. The above steps are repeated on another `QueueTest` object, this time running the `DequeueWorks` test.

Availability: Linux, Windows, Mac.

Invoking the Tests

`TEST()` and `TEST_F()` implicitly register their tests with googletest. So, unlike with many other C++ testing frameworks, 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.

When invoked, the `RUN_ALL_TESTS()` macro:

- Saves the state of all googletest flags.
- Creates a test fixture object for the first test.
- Initializes it via `SetUp()`.
- Runs the test on the fixture object.
- Cleans up the fixture via `TearDown()`.
- Deletes the fixture.
- Restores the state of all googletest flags.
- Repeats the above steps for the next test, until all tests have run.

If a fatal failure happens the subsequent steps will be skipped.

{: .callout .important}

> IMPORTANT: You must **not** ignore the return value of `RUN_ALL_TESTS()`, or you will get a compiler error. The rationale for this design is that the automated testing service determines whether a test has passed based on its exit code, not on its stdout/stderr output; thus your `main()` function must return the value of `RUN_ALL_TESTS()`.
>
> Also, you should call `RUN_ALL_TESTS()` only **once**. Calling it more than once conflicts with some advanced googletest features (e.g., thread-safe death tests) and thus is not supported.

Availability: Linux, Windows, Mac.

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. See the end of this section for details. The remainder of this section should only apply when you need to do something custom before the tests run that cannot be expressed within the framework of fixtures and test suites.

If you write your own `main` function, it should return the value of `RUN_ALL_TESTS()`.

You can start from this boilerplate:

```
#include "this/package/foo.h"

#include "gtest/gtest.h"

namespace my {
namespace project {
namespace {

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {
 protected:
  // You can remove any or all of the following functions if their bodies would
  // be empty.

  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.
}

}  // namespace
}  // namespace project
}  // namespace my
```

```
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

The `::testing::InitGoogleTest()` function parses the command line for googletest flags, and removes all recognized flags. This allows the user to control a test program's behavior via various flags, which we'll cover in the AdvancedGuide. You **must** call this function before calling `RUN_ALL_TESTS()`, or the flags won't be properly initialized.

On Windows, `InitGoogleTest()` also works with wide strings, so it can be used in programs compiled in `UNICODE` mode as well.

But maybe you think that writing all those `main` functions is too much work? We agree with you completely, and that's why Google Test provides a basic implementation of main(). If it fits your needs, then just link your test with the `gtest_main` library and you are good to go.

{: .callout .note}
NOTE: `ParseGUnitFlags()` is deprecated in favor of `InitGoogleTest()`.

Known Limitations

- Google Test is designed to be thread-safe. The implementation is thread-safe on systems where the `pthreads` library is available. It is currently *unsafe* to use Google Test assertions from two threads concurrently on other systems (e.g. Windows). In most tests this is not an issue as usually the assertions are done in the main thread. If you want to help, you can volunteer to implement the necessary synchronization primitives in `gtest-port.h` for your platform.