# Introduction: Why Google C++ Testing Framework?

*Google C++ Testing Framework* helps you write better C++ tests.

No matter whether you work on Linux, Windows, or a Mac, if you write C++ code, Google Test can help you.

So what makes a good test, and how does Google C++ Testing Framework 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. Google C++ Testing Framework isolates the tests by running each of them on a different object. When a test fails, Google C++ Testing Framework allows you to run it in isolation for quick debugging.
2. Tests should be well *organized* and reflect the structure of the tested code. Google C++ Testing Framework groups related tests into test cases 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*. The open-source community has a lot of code that is platform-neutral, its tests should also be platform-neutral. Google C++ Testing Framework works on different OSes, with different compilers (gcc, MSVC, and others), with or without exceptions, so Google C++ Testing Framework tests can easily work with a variety of configurations. (Note that the current release only contains build scripts for Linux - we are actively working on scripts for other platforms.)
4. When tests fail, they should provide as much *information* about the problem as possible. Google C++ Testing Framework 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*. Google C++ Testing Framework 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 Google C++ Testing Framework, 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 Google C++ Testing Framework 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!

*Note:* We sometimes refer to Google C++ Testing Framework informally as *Google Test*.

# Setting up a New Test Project

To write a test program using Google Test, you need to compile Google Test into a library and link your test with it. We provide build files for some popular build systems: `msvc/` for Visual Studio, `xcode/` for Mac Xcode, `make/` for GNU make, `codegear/` for Borland C++ Builder, and the autotools script (deprecated) and `CMakeLists.txt` for CMake (recommended) in the Google Test root directory. If your build system is not on this list, you can take a look at `make/Makefile` to learn how Google Test should be compiled (basically you want to compile `src/gtest-all.cc` with `GTEST_ROOT` and `GTEST_ROOT/include` in the header search path, where `GTEST_ROOT` is the Google Test root directory).

Once you are able to compile the Google Test library, you should create a project or build target for your test program. Make sure you have `GTEST_ROOT/include` in the header search path so that the compiler can find `"gtest/gtest.h"` when compiling your test. Set up your test project to link with the Google Test library (for example, in Visual Studio, this is done by adding a dependency on `gtest.vcproj`).

If you still have questions, take a look at how Google Test's own tests are built and use them as examples.

# Basic Concepts

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

A *test program* can contain multiple test cases.

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

# Assertions

Google Test assertions are macros that resemble function calls. You test a class or function by making assertions about its behavior. When an assertion fails, Google Test 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 Google Test'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 failures 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. An 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;
}
```

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.

## Basic Assertions

These assertions do basic true/false condition testing.

| | | |
|---|---|---|

| Fatal assertion | Nonfatal assertion | Verifies |
| --- | --- | --- |
| `ASSERT_TRUE(`*condition*`);` | `EXPECT_TRUE(`*condition*`);` | *condition* is true |
| `ASSERT_FALSE(`*condition*`);` | `EXPECT_FALSE(`*condition*`);` | *condition* is false |

Remember, when they fail, `ASSERT_*` yields a fatal failure and returns from the current function, while `EXPECT_*` yields a nonfatal failure, allowing the function to continue running. In either case, an assertion failure means its containing test fails.

*Availability*: Linux, Windows, Mac.

## Binary Comparison

This section describes assertions that compare two values.

| Fatal assertion | Nonfatal assertion | Verifies |
| --- | --- | --- |
| `ASSERT_EQ(`*val1*`,`*val2*`);` | `EXPECT_EQ(`*val1*`,`*val2*`);` | *val1* `==` *val2* |
| `ASSERT_NE(`*val1*`,`*val2*`);` | `EXPECT_NE(`*val1*`,`*val2*`);` | *val1* `!=` *val2* |
| `ASSERT_LT(`*val1*`,`*val2*`);` | `EXPECT_LT(`*val1*`,`*val2*`);` | *val1* `<` *val2* |
| `ASSERT_LE(`*val1*`,`*val2*`);` | `EXPECT_LE(`*val1*`,`*val2*`);` | *val1* `<=` *val2* |
| `ASSERT_GT(`*val1*`,`*val2*`);` | `EXPECT_GT(`*val1*`,`*val2*`);` | *val1* `>` *val2* |
| `ASSERT_GE(`*val1*`,`*val2*`);` | `EXPECT_GE(`*val1*`,`*val2*`);` | *val1* `>=` *val2* |

In the event of a failure, Google Test prints both *val1* and *val2*.

Value arguments must be comparable by the assertion's comparison operator or you'll get a compiler error. We used to require the arguments to support the `<<` operator for streaming to an `ostream`, but it's no longer necessary since v1.6.0 (if `<<` is supported, it will be called to print the arguments when the assertion fails; otherwise Google Test will attempt to print them in the best way it can. For more details and how to customize the printing of the arguments, see this Google Mock recipe.).

These assertions can work with a user-defined type, but only if you define the corresponding comparison operator (e.g. `==`, `<`, etc). If the corresponding operator is defined, prefer using the `ASSERT_*()` macros because they will print out not only the result of the comparison, but the two operands as well.

Arguments are always evaluated exactly once. Therefore, it's OK for the arguments to have side effects. However, as with any ordinary C/C++ function, the arguments' evaluation order is undefined (i.e. the compiler is free to choose

any order) and your code should not depend on any particular argument evaluation order.

`ASSERT_EQ()` does pointer equality on pointers. If used on two C strings, it tests if they are in the same memory location, not if they have the same value. Therefore, if you want to compare C strings (e.g. `const char*`) by value, use `ASSERT_STREQ()`, which will be described later on. In particular, to assert that a C string is `NULL`, use `ASSERT_STREQ(NULL, c_string)`. However, to compare two `string` objects, you should use `ASSERT_EQ`.

Macros in this section work with both narrow and wide string objects (`string` and `wstring`).

*Availability*: Linux, Windows, Mac.

*Historical note*: Before February 2016 `*_EQ` had a convention of calling it as `ASSERT_EQ(expected, actual)`, so lots of existing code uses this order. Now `*_EQ` treats both parameters in the same way.

## String Comparison

The assertions in this group compare two **C strings**. If you want to compare two `string` objects, use `EXPECT_EQ`, `EXPECT_NE`, and etc instead.

| Fatal assertion | Nonfatal assertion | |
|---|---|---|
| `ASSERT_STREQ(_str1_,_str2_);` | `EXPECT_STREQ(_str1_,_str_2);` | |
| `ASSERT_STRNE(_str1_,_str2_);` | `EXPECT_STRNE(_str1_,_str2_);` | |
| `ASSERT_STRCASEEQ(_str1_,_str2_);` | `EXPECT_STRCASEEQ(_str1_,_str2_);` | |
| `ASSERT_STRCASENE(_str1_,_str2_);` | `EXPECT_STRCASENE(_str1_,_str2_);` | |

| | | |
|---|---|---|
| | | |

Note that "CASE" in an assertion name means that case is ignored.

`*STREQ*` and `*STRNE*` also accept wide C strings ( `wchar_t*` ). If a comparison of two wide strings fails, their values will be printed as UTF-8 narrow strings.

A `NULL` pointer and an empty string are considered *different*.

*Availability*: Linux, Windows, Mac.

See also: For more string comparison tricks (substring, prefix, suffix, and regular expression matching, for example), see the Advanced Google Test Guide.

# 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 Google Test 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(test_case_name, test_name) {
 ... test body ...
}
```

`TEST()` arguments go from general to specific. The *first* argument is the name of the test case, and the *second* argument is the test's name within the test case. Both names must be valid C++ identifiers, and they should not contain underscore ( `_` ). A test's *full name* consists of its containing test case and its individual name. Tests from different test cases 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 case for this function might look like:

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

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(1, Factorial(1));
```

```
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

Google Test groups the test results by test cases, so logically-related tests should be in the same test case; 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 case `FactorialTest`.

*Availability*: Linux, Windows, Mac.

# Test Fixtures: Using the Same Data Configuration for Multiple Tests

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

To create a fixture, just:

1. Derive a class from `::testing::Test`. Start its body with `protected:` or `public:` 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` - don't let that happen to you.
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 this FAQ entry.
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(test_case_name, test_name) {
 ... test body ...
}
```

Like `TEST()`, the first argument is the test case 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()`, Google Test will:

1. Create a *fresh* test fixture at runtime
2. Immediately initialize it via `SetUp()`,
3. Run the test
4. Clean up by calling `TearDown()`

5. Delete the test fixture. Note that different tests in the same test case have different test fixture objects, and Google Test always deletes a test fixture before it creates the next one. Google Test 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:
  virtual void SetUp() {
    q1_.Enqueue(1);
    q2_.Enqueue(2);
    q2_.Enqueue(3);
  }

  // virtual void TearDown() {}

  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(0, q0_.size());
}

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

  n = q1_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(1, *n);
  EXPECT_EQ(0, q1_.size());
  delete n;
```

```
  n = q2_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(2, *n);
  EXPECT_EQ(1, q2_.size());
  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_TRUE(n != NULL)`, 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. Google Test 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.

*Note*: Google Test automatically saves all *Google Test* flags when a test object is constructed, and restores them when it is destructed.

# Invoking the Tests

`TEST()` and `TEST_F()` implicitly register their tests with Google Test. 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 cases, or even different source files.

When invoked, the `RUN_ALL_TESTS()` macro:

1. Saves the state of all Google Test flags.
2. Creates a test fixture object for the first test.
3. Initializes it via `SetUp()`.
4. Runs the test on the fixture object.
5. Cleans up the fixture via `TearDown()`.
6. Deletes the fixture.
7. Restores the state of all Google Test flags.
8. Repeats the above steps for the next test, until all tests have run.

In addition, if the text fixture's constructor generates a fatal failure in step 2, there is no point for step 3 - 5 and they are thus skipped. Similarly, if step 3 generates a fatal failure, step 4 will be skipped.

*Important*: You must not ignore the return value of `RUN_ALL_TESTS()`, or `gcc` will give you 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 Google Test features (e.g. thread-safe death tests) and thus is not supported.

*Availability*: Linux, Windows, Mac.

# Writing the main() Function

You can start from this boilerplate:

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

namespace {

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {
 protected:
  // You can remove any or all of the following functions if its body
  // is empty.

  FooTest() {
    // You can do set-up work for each test here.
  }

  virtual ~FooTest() {
    // 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:

  virtual void SetUp() {
    // Code here will be called immediately after the constructor (right
    // before each test).
  }

  virtual void TearDown() {
    // Code here will be called immediately after each test (right
    // before the destructor).
  }

  // Objects declared here can be used by all tests in the test case for Foo.
};

// Tests that the Foo::Bar() method does Abc.
TEST_F(FooTest, MethodBarDoesAbc) {
  const string input_filepath = "this/package/testdata/myinputfile.dat";
  const string output_filepath = "this/package/testdata/myoutputfile.dat";
  Foo f;
  EXPECT_EQ(0, f.Bar(input_filepath, output_filepath));
}
```

```
// Tests that Foo does Xyz.
TEST_F(FooTest, DoesXyz) {
  // Exercises the Xyz feature of Foo.
}

}  // namespace

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

The `::testing::InitGoogleTest()` function parses the command line for Google Test 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 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 gtest_main library and you are good to go.

## Important note for Visual C++ users

If you put your tests into a library and your `main()` function is in a different library or in your .exe file, those tests will not run. The reason is a bug in Visual C++. When you define your tests, Google Test creates certain static objects to register them. These objects are not referenced from elsewhere but their constructors are still supposed to run. When Visual C++ linker sees that nothing in the library is referenced from other places it throws the library out. You have to reference your library with tests from your main program to keep the linker from discarding it. Here is how to do it. Somewhere in your library code declare a function:

```
__declspec(dllexport) int PullInMyLibrary() { return 0; }
```

If you put your tests in a static library (not DLL) then `__declspec(dllexport)` is not required. Now, in your main program, write a code that invokes that function:

```
int PullInMyLibrary();
static int dummy = PullInMyLibrary();
```

This will keep your tests referenced and will make them register themselves at startup.

In addition, if you define your tests in a static library, add `/OPT:NOREF` to your main program linker options. If you use MSVC++ IDE, go to your .exe project properties/Configuration Properties/Linker/Optimization and set References setting to `Keep Unreferenced Data (/OPT:NOREF)`. This will keep Visual C++ linker from discarding individual symbols generated by your tests from the final executable.

There is one more pitfall, though. If you use Google Test as a static library (that's how it is defined in gtest.vcproj) your tests must also reside in a static library. If you have to have them in a DLL, you *must* change Google Test to build into a DLL as well. Otherwise your tests will not run correctly or will not run at all. The general conclusion here is: make your life easier - do not write your tests in libraries!

# Where to Go from Here

Congratulations! You've learned the Google Test basics. You can start writing and running Google Test tests, read some samples, or continue with AdvancedGuide, which describes many more useful Google Test features.

# 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.