GoogleTest FAQ

Why should test suite names and test names not contain underscore?

{: .callout .note}
Note: GoogleTest reserves underscore (`_`) for special purpose keywords, such as the `DISABLED_` prefix, in addition
to the following rationale.

Underscore (`_`) is special, as C++ reserves the following to be used by the compiler and the standard library:

1. any identifier that starts with an `_` followed by an upper-case letter, and
2. any identifier that contains two consecutive underscores (i.e. `__`) *anywhere* in its name.

User code is *prohibited* from using such identifiers.

Now let's look at what this means for `TEST` and `TEST_F`.

Currently `TEST(TestSuiteName, TestName)` generates a class named `TestSuiteName_TestName_Test`. What happens if `TestSuiteName` or `TestName` contains `_`?

1. If `TestSuiteName` starts with an `_` followed by an upper-case letter (say, `_Foo`), we end up with `_Foo_TestName_Test`, which is reserved and thus invalid.
2. If `TestSuiteName` ends with an `_` (say, `Foo_`), we get `Foo__TestName_Test`, which is invalid.
3. If `TestName` starts with an `_` (say, `_Bar`), we get `TestSuiteName__Bar_Test`, which is invalid.
4. If `TestName` ends with an `_` (say, `Bar_`), we get `TestSuiteName_Bar__Test`, which is invalid.

So clearly `TestSuiteName` and `TestName` cannot start or end with `_` (Actually, `TestSuiteName` can start with `_` -- as long as the `_` isn't followed by an upper-case letter. But that's getting complicated. So for simplicity we just say that it cannot start with `_`.).

It may seem fine for `TestSuiteName` and `TestName` to contain `_` in the middle. However, consider this:

```
TEST(Time, Flies_Like_An_Arrow) { ... }
TEST(Time_Flies, Like_An_Arrow) { ... }
```

Now, the two `TEST`s will both generate the same class (`Time_Flies_Like_An_Arrow_Test`). That's not good.

So for simplicity, we just ask the users to avoid `_` in `TestSuiteName` and `TestName`. The rule is more constraining than necessary, but it's simple and easy to remember. It also gives GoogleTest some wiggle room in case its implementation needs to change in the future.

If you violate the rule, there may not be immediate consequences, but your test may (just may) break with a new compiler (or a new version of the compiler you are using) or with a new version of GoogleTest. Therefore it's best to follow the rule.

Why does GoogleTest support `EXPECT_EQ(NULL, ptr)` and `ASSERT_EQ(NULL, ptr)` but not `EXPECT_NE(NULL, ptr)` and `ASSERT_NE(NULL, ptr)`?

First of all, you can use `nullptr` with each of these macros, e.g. `EXPECT_EQ(ptr, nullptr)`, `EXPECT_NE(ptr, nullptr)`, `ASSERT_EQ(ptr, nullptr)`, `ASSERT_NE(ptr, nullptr)`. This is the preferred syntax in the style guide because `nullptr` does not have the type problems that `NULL` does.

Due to some peculiarity of C++, it requires some non-trivial template meta programming tricks to support using `NULL` as an argument of the `EXPECT_XX()` and `ASSERT_XX()` macros. Therefore we only do it where it's most needed (otherwise we make the implementation of GoogleTest harder to maintain and more error-prone than necessary).

Historically, the `EXPECT_EQ()` macro took the *expected* value as its first argument and the *actual* value as the second, though this argument order is now discouraged. It was reasonable that someone wanted to write `EXPECT_EQ(NULL, some_expression)`, and this indeed was requested several times. Therefore we implemented it.

The need for `EXPECT_NE(NULL, ptr)` wasn't nearly as strong. When the assertion fails, you already know that `ptr` must be `NULL`, so it doesn't add any information to print `ptr` in this case. That means `EXPECT_TRUE(ptr != NULL)` works just as well.

If we were to support `EXPECT_NE(NULL, ptr)`, for consistency we'd have to support `EXPECT_NE(ptr, NULL)` as well. This means using the template meta programming tricks twice in the implementation, making it even harder to understand and maintain. We believe the benefit doesn't justify the cost.

Finally, with the growth of the gMock matcher library, we are encouraging people to use the unified `EXPECT_THAT(value, matcher)` syntax more often in tests. One significant advantage of the matcher approach is that matchers can be easily combined to form new matchers, while the `EXPECT_NE`, etc, macros cannot be easily combined. Therefore we want to invest more in the matchers than in the `EXPECT_XX()` macros.

I need to test that different implementations of an interface satisfy some common requirements. Should I use typed tests or value-parameterized tests?

For testing various implementations of the same interface, either typed tests or value-parameterized tests can get it done. It's really up to you the user to decide which is more convenient for you, depending on your particular case. Some rough guidelines:

- Typed tests can be easier to write if instances of the different implementations can be created the same way, modulo the type. For example, if all these implementations have a public default constructor (such that you can write `new TypeParam`), or if their factory functions have the same form (e.g. `CreateInstance<TypeParam>()`).
- Value-parameterized tests can be easier to write if you need different code patterns to create different implementations' instances, e.g. `new Foo` vs `new Bar(5)`. To accommodate for the differences, you can write factory function wrappers and pass these function pointers to the tests as their parameters.
- When a typed test fails, the default output includes the name of the type, which can help you quickly identify which implementation is wrong. Value-parameterized tests only show the number of the failed iteration by default. You will need to define a function that returns the iteration name and pass it as the third parameter to INSTANTIATE_TEST_SUITE_P to have more useful output.
- When using typed tests, you need to make sure you are testing against the interface type, not the concrete types (in other words, you want to make sure `implicit_cast<MyInterface*>(my_concrete_impl)` works, not just that `my_concrete_impl` works). It's less likely to make mistakes in this area when using value-parameterized tests.

I hope I didn't confuse you more. :-) If you don't mind, I'd suggest you to give both approaches a try. Practice is a much better way to grasp the subtle differences between the two tools. Once you have some concrete experience, you can much more easily decide which one to use the next time.

I got some run-time errors about invalid proto descriptors when using `ProtocolMessageEquals`. Help!

{: .callout .note}
Note: `ProtocolMessageEquals` and `ProtocolMessageEquiv` are *deprecated* now. Please use `EqualsProto`, etc instead.

`ProtocolMessageEquals` and `ProtocolMessageEquiv` were redefined recently and are now less tolerant of invalid protocol buffer definitions. In particular, if you have a `foo.proto` that doesn't fully qualify the type of a protocol message it references (e.g. `message<Bar>` where it should be `message<blah.Bar>`), you

will now get run-time errors like:

```
... descriptor.cc:...] Invalid proto descriptor for file "path/to/foo.proto":
... descriptor.cc:...]  blah.MyMessage.my_field: ".Bar" is not defined.
```

If you see this, your `.proto` file is broken and needs to be fixed by making the types fully qualified. The new definition of `ProtocolMessageEquals` and `ProtocolMessageEquiv` just happen to reveal your bug.

My death test modifies some state, but the change seems lost after the death test finishes. Why?

Death tests (`EXPECT_DEATH`, etc) are executed in a sub-process s.t. the expected crash won't kill the test program (i.e. the parent process). As a result, any in-memory side effects they incur are observable in their respective sub-processes, but not in the parent process. You can think of them as running in a parallel universe, more or less.

In particular, if you use mocking and the death test statement invokes some mock methods, the parent process will think the calls have never occurred. Therefore, you may want to move your `EXPECT_CALL` statements inside the `EXPECT_DEATH` macro.

EXPECT_EQ(htonl(blah), blah_blah) generates weird compiler errors in opt mode. Is this a GoogleTest bug?

Actually, the bug is in `htonl()`.

According to `'man htonl'`, `htonl()` is a *function*, which means it's valid to use `htonl` as a function pointer. However, in opt mode `htonl()` is defined as a *macro*, which breaks this usage.

Worse, the macro definition of `htonl()` uses a `gcc` extension and is *not* standard C++. That hacky implementation has some ad hoc limitations. In particular, it prevents you from writing `Foo<sizeof(htonl(x))>()`, where `Foo` is a template that has an integral argument.

The implementation of `EXPECT_EQ(a, b)` uses `sizeof(... a ...)` inside a template argument, and thus doesn't compile in opt mode when `a` contains a call to `htonl()`. It is difficult to make `EXPECT_EQ` bypass the `htonl()` bug, as the solution must work with different compilers on various platforms.

The compiler complains about "undefined references" to some static const member variables, but I did define them in the class body. What's wrong?

If your class has a static data member:

```
// foo.h
class Foo {
  ...
  static const int kBar = 100;
};
```

You also need to define it *outside* of the class body in `foo.cc`:

```
const int Foo::kBar;  // No initializer here.
```

Otherwise your code is **invalid C++**, and may break in unexpected ways. In particular, using it in GoogleTest comparison assertions (`EXPECT_EQ`, etc) will generate an "undefined reference" linker error. The fact that "it used to work" doesn't mean it's valid. It just means that you were lucky. :-)

If the declaration of the static data member is `constexpr` then it is implicitly an `inline` definition, and a separate definition in `foo.cc` is not needed:

```
// foo.h
class Foo {
  ...
  static constexpr int kBar = 100;  // Defines kBar, no need to do it in foo.cc.
};
```

Can I derive a test fixture from another?

Yes.

Each test fixture has a corresponding and same named test suite. This means only one test suite can use a particular fixture. Sometimes, however, multiple test cases may want to use the same or slightly different fixtures. For example, you may want to make sure that all of a GUI library's test suites don't leak important system resources like fonts and brushes.

In GoogleTest, you share a fixture among test suites by putting the shared logic in a base test fixture, then deriving from that base a separate fixture for each test suite that wants to use this common logic. You then use `TEST_F()` to write tests using each derived fixture.

Typically, your code looks like this:

```
// Defines a base test fixture.
class BaseTest : public ::testing::Test {
 protected:
  ...
};

// Derives a fixture FooTest from BaseTest.
class FooTest : public BaseTest {
 protected:
  void SetUp() override {
    BaseTest::SetUp();  // Sets up the base fixture first.
```

```
    ... additional set-up work ...
  }

  void TearDown() override {
    ... clean-up work for FooTest ...
    BaseTest::TearDown();  // Remember to tear down the base fixture
                           // after cleaning up FooTest!
  }

  ... functions and variables for FooTest ...
};

// Tests that use the fixture FooTest.
TEST_F(FooTest, Bar) { ... }
TEST_F(FooTest, Baz) { ... }

... additional fixtures derived from BaseTest ...
```

If necessary, you can continue to derive test fixtures from a derived fixture. GoogleTest has no limit on how deep the hierarchy can be.

For a complete example using derived test fixtures, see sample5_unittest.cc.

My compiler complains "void value not ignored as it ought to be." What does this mean?

You're probably using an `ASSERT_*()` in a function that doesn't return `void`. `ASSERT_*()` can only be used in `void` functions, due to exceptions being disabled by our build system. Please see more details here.

My death test hangs (or seg-faults). How do I fix it?

In GoogleTest, death tests are run in a child process and the way they work is delicate. To write death tests you really need to understand how they work—see the details at Death Assertions in the Assertions Reference.

In particular, death tests don't like having multiple threads in the parent process. So the first thing you can try is to eliminate creating threads outside of `EXPECT_DEATH()`. For example, you may want to use mocks or fake objects instead of real ones in your tests.

Sometimes this is impossible as some library you must use may be creating threads before `main()` is even reached. In this case, you can try to minimize the chance of conflicts by either moving as many activities as possible inside `EXPECT_DEATH()` (in the extreme case, you want to move everything inside), or leaving as few things as possible in it. Also, you can try to set the death test style to `"threadsafe"`, which is safer but slower, and see if it helps.

If you go with thread-safe death tests, remember that they rerun the test program from the beginning in the child process. Therefore make sure your program can run side-by-side with itself and is deterministic.

In the end, this boils down to good concurrent programming. You have to make sure that there are no race conditions or deadlocks in your program. No silver bullet - sorry!

Should I use the constructor/destructor of the test fixture or SetUp()/TearDown()? {#CtorVsSetUp}

The first thing to remember is that GoogleTest does **not** reuse the same test fixture object across multiple tests. For each `TEST_F`, GoogleTest will create a **fresh** test fixture object, immediately call `SetUp()`, run the test body, call `TearDown()`, and then delete the test fixture object.

When you need to write per-test set-up and tear-down logic, you have the choice between using the test fixture constructor/destructor or `SetUp()/TearDown()`. The former is usually preferred, as it has the following benefits:

- By initializing a member variable in the constructor, we have the option to make it `const`, which helps prevent accidental changes to its value and makes the tests more obviously correct.
- In case we need to subclass the test fixture class, the subclass' constructor is guaranteed to call the base class' constructor *first*, and the subclass' destructor is guaranteed to call the base class' destructor *afterward*. With `SetUp()/TearDown()`, a subclass may make the mistake of forgetting to call the base class' `SetUp()/TearDown()` or call them at the wrong time.

You may still want to use `SetUp()/TearDown()` in the following cases:

- C++ does not allow virtual function calls in constructors and destructors. You can call a method declared as virtual, but it will not use dynamic dispatch. It will use the definition from the class the constructor of which is currently executing. This is because calling a virtual method before the derived class constructor has a chance to run is very dangerous - the virtual method might operate on uninitialized data. Therefore, if you need to call a method that will be overridden in a derived class, you have to use `SetUp()/TearDown()`.
- In the body of a constructor (or destructor), it's not possible to use the `ASSERT_xx` macros. Therefore, if the set-up operation could cause a fatal test failure that should prevent the test from running, it's necessary to use `abort` and abort the whole test executable, or to use `SetUp()` instead of a constructor.
- If the tear-down operation could throw an exception, you must use `TearDown()` as opposed to the destructor, as throwing in a destructor leads to undefined behavior and usually will kill your program right away. Note that many standard libraries (like STL) may throw when exceptions are enabled in the compiler. Therefore you should prefer `TearDown()` if you want to write portable tests that work with or without exceptions.
- The GoogleTest team is considering making the assertion macros throw on platforms where exceptions are enabled (e.g. Windows, Mac OS, and Linux client-side), which will eliminate the need for the user to propagate failures from a subroutine to its caller. Therefore, you shouldn't use

GoogleTest assertions in a destructor if your code could run on such a platform.

The compiler complains "no matching function to call" when I use ASSERT_PRED*. How do I fix it?

See details for `EXPECT_PRED*` in the Assertions Reference.

My compiler complains about "ignoring return value" when I call RUN_ALL_TESTS(). Why?

Some people had been ignoring the return value of `RUN_ALL_TESTS()`. That is, instead of

```
return RUN_ALL_TESTS();
```

they write

```
RUN_ALL_TESTS();
```

This is **wrong and dangerous**. The testing services needs to see the return value of `RUN_ALL_TESTS()` in order to determine if a test has passed. If your `main()` function ignores it, your test will be considered successful even if it has a GoogleTest assertion failure. Very bad.

We have decided to fix this (thanks to Michael Chastain for the idea). Now, your code will no longer be able to ignore `RUN_ALL_TESTS()` when compiled with `gcc`. If you do so, you'll get a compiler error.

If you see the compiler complaining about you ignoring the return value of `RUN_ALL_TESTS()`, the fix is simple: just make sure its value is used as the return value of `main()`.

But how could we introduce a change that breaks existing tests? Well, in this case, the code was already broken in the first place, so we didn't break it. :-)

My compiler complains that a constructor (or destructor) cannot return a value. What's going on?

Due to a peculiarity of C++, in order to support the syntax for streaming messages to an `ASSERT_*`, e.g.

```
ASSERT_EQ(1, Foo()) << "blah blah" << foo;
```

we had to give up using `ASSERT*` and `FAIL*` (but not `EXPECT*` and `ADD_FAILURE*`) in constructors and destructors. The workaround is to move the content of your constructor/destructor to a private void member function, or switch to `EXPECT_*()` if that works. This section in the user's guide explains it.

My SetUp() function is not called. Why?

C++ is case-sensitive. Did you spell it as `Setup()`?

Similarly, sometimes people spell `SetUpTestSuite()` as `SetupTestSuite()` and wonder why it's never called.

I have several test suites which share the same test fixture logic, do I have to define a new test fixture class for each of them? This seems pretty tedious.

You don't have to. Instead of

```
class FooTest : public BaseTest {};

TEST_F(FooTest, Abc) { ... }
TEST_F(FooTest, Def) { ... }

class BarTest : public BaseTest {};

TEST_F(BarTest, Abc) { ... }
TEST_F(BarTest, Def) { ... }
```

you can simply `typedef` the test fixtures:

```
typedef BaseTest FooTest;

TEST_F(FooTest, Abc) { ... }
TEST_F(FooTest, Def) { ... }

typedef BaseTest BarTest;

TEST_F(BarTest, Abc) { ... }
TEST_F(BarTest, Def) { ... }
```

GoogleTest output is buried in a whole bunch of LOG messages. What do I do?

The GoogleTest output is meant to be a concise and human-friendly report. If your test generates textual output itself, it will mix with the GoogleTest output, making it hard to read. However, there is an easy solution to this problem.

Since `LOG` messages go to stderr, we decided to let GoogleTest output go to stdout. This way, you can easily separate the two using redirection. For example:

```
$ ./my_test > gtest_output.txt
```

Why should I prefer test fixtures over global variables?

There are several good reasons:

1. It's likely your test needs to change the states of its global variables. This makes it difficult to keep side effects from escaping one test and contaminating others, making debugging difficult. By using fixtures, each test has a fresh set of variables that's different (but with the same names). Thus, tests are kept independent of each other.
2. Global variables pollute the global namespace.
3. Test fixtures can be reused via subclassing, which cannot be done easily with global variables. This is useful if many test suites have something in common.

What can the statement argument in ASSERT_DEATH() be?

`ASSERT_DEATH(statement, matcher)` (or any death assertion macro) can be used wherever `statement` is valid. So basically `statement` can be any C++ statement that makes sense in the current context. In particular, it can reference global and/or local variables, and can be:

- a simple function call (often the case),
- a complex expression, or
- a compound statement.

Some examples are shown here:

```
// A death test can be a simple function call.
TEST(MyDeathTest, FunctionCall) {
  ASSERT_DEATH(Xyz(5), "Xyz failed");
}

// Or a complex expression that references variables and functions.
TEST(MyDeathTest, ComplexExpression) {
  const bool c = Condition();
  ASSERT_DEATH((c ? Func1(0) : object2.Method("test")),
               "(Func1|Method) failed");
}

// Death assertions can be used anywhere in a function.  In
// particular, they can be inside a loop.
TEST(MyDeathTest, InsideLoop) {
  // Verifies that Foo(0), Foo(1), ..., and Foo(4) all die.
  for (int i = 0; i < 5; i++) {
    EXPECT_DEATH_M(Foo(i), "Foo has \\d+ errors",
                   ::testing::Message() << "where i is " << i);
  }
}

// A death assertion can contain a compound statement.
TEST(MyDeathTest, CompoundStatement) {
  // Verifies that at lease one of Bar(0), Bar(1), ..., and
  // Bar(4) dies.
  ASSERT_DEATH({
    for (int i = 0; i < 5; i++) {
```

```
      Bar(i);
    }
  },
  "Bar has \\d+ errors");
}
```

I have a fixture class `FooTest`, but `TEST_F(FooTest, Bar)` gives me error `"no matching function for call to `FooTest::FooTest()'"`. Why?

GoogleTest needs to be able to create objects of your test fixture class, so it must have a default constructor. Normally the compiler will define one for you. However, there are cases where you have to define your own:

- If you explicitly declare a non-default constructor for class `FooTest` (`DISALLOW_EVIL_CONSTRUCTORS()` does this), then you need to define a default constructor, even if it would be empty.
- If `FooTest` has a const non-static data member, then you have to define the default constructor *and* initialize the const member in the initializer list of the constructor. (Early versions of `gcc` doesn't force you to initialize the const member. It's a bug that has been fixed in `gcc 4`.)

Why does ASSERT_DEATH complain about previous threads that were already joined?

With the Linux pthread library, there is no turning back once you cross the line from a single thread to multiple threads. The first time you create a thread, a manager thread is created in addition, so you get 3, not 2, threads. Later when the thread you create joins the main thread, the thread count decrements by 1, but the manager thread will never be killed, so you still have 2 threads, which means you cannot safely run a death test.

The new NPTL thread library doesn't suffer from this problem, as it doesn't create a manager thread. However, if you don't control which machine your test runs on, you shouldn't depend on this.

Why does GoogleTest require the entire test suite, instead of individual tests, to be named *DeathTest when it uses ASSERT_DEATH?

GoogleTest does not interleave tests from different test suites. That is, it runs all tests in one test suite first, and then runs all tests in the next test suite, and so on. GoogleTest does this because it needs to set up a test suite before the first test in it is run, and tear it down afterwards. Splitting up the test case would require multiple set-up and tear-down processes, which is inefficient and makes the semantics unclean.

If we were to determine the order of tests based on test name instead of test case name, then we would have a problem with the following situation:

```
TEST_F(FooTest, AbcDeathTest) { ... }
TEST_F(FooTest, Uvw) { ... }

TEST_F(BarTest, DefDeathTest) { ... }
TEST_F(BarTest, Xyz) { ... }
```

Since `FooTest.AbcDeathTest` needs to run before `BarTest.Xyz`, and we don't interleave tests from different test suites, we need to run all tests in the `FooTest` case before running any test in the `BarTest` case. This contradicts with the requirement to run `BarTest.DefDeathTest` before `FooTest.Uvw`.

But I don't like calling my entire test suite *DeathTest when it contains both death tests and non-death tests. What do I do?

You don't have to, but if you like, you may split up the test suite into `FooTest` and `FooDeathTest`, where the names make it clear that they are related:

```
class FooTest : public ::testing::Test { ... };

TEST_F(FooTest, Abc) { ... }
TEST_F(FooTest, Def) { ... }

using FooDeathTest = FooTest;

TEST_F(FooDeathTest, Uvw) { ... EXPECT_DEATH(...) ... }
TEST_F(FooDeathTest, Xyz) { ... ASSERT_DEATH(...) ... }
```

GoogleTest prints the LOG messages in a death test's child process only when the test fails. How can I see the LOG messages when the death test succeeds?

Printing the LOG messages generated by the statement inside `EXPECT_DEATH()` makes it harder to search for real problems in the parent's log. Therefore, GoogleTest only prints them when the death test has failed.

If you really need to see such LOG messages, a workaround is to temporarily break the death test (e.g. by changing the regex pattern it is expected to match). Admittedly, this is a hack. We'll consider a more permanent solution after the fork-and-exec-style death tests are implemented.

The compiler complains about `no match for 'operator<<'` when I use an assertion. What gives?

If you use a user-defined type `FooType` in an assertion, you must make sure there is an `std::ostream& operator<<(std::ostream&, const FooType&)` function defined such that we can print a value of `FooType`.

In addition, if `FooType` is declared in a name space, the `<<` operator also needs to be defined in the *same* name space. See Tip of the Week #49 for details.

How do I suppress the memory leak messages on Windows?

Since the statically initialized GoogleTest singleton requires allocations on the heap, the Visual C++ memory leak detector will report memory leaks at the end of the program run. The easiest way to avoid this is to use the `_CrtMemCheckpoint` and `_CrtMemDumpAllObjectsSince` calls to not report any statically initialized heap objects. See MSDN for more details and additional heap check/debug routines.

How can my code detect if it is running in a test?

If you write code that sniffs whether it's running in a test and does different things accordingly, you are leaking test-only logic into production code and there is no easy way to ensure that the test-only code paths aren't run by mistake in production. Such cleverness also leads to Heisenbugs. Therefore we strongly advise against the practice, and GoogleTest doesn't provide a way to do it.

In general, the recommended way to cause the code to behave differently under test is Dependency Injection. You can inject different functionality from the test and from the production code. Since your production code doesn't link in the for-test logic at all (the `testonly` attribute for BUILD targets helps to ensure that), there is no danger in accidentally running it.

However, if you *really*, *really*, *really* have no choice, and if you follow the rule of ending your test program names with `_test`, you can use the *horrible* hack of sniffing your executable name (`argv[0]` in `main()`) to know whether the code is under test.

How do I temporarily disable a test?

If you have a broken test that you cannot fix right away, you can add the `DISABLED_` prefix to its name. This will exclude it from execution. This is better than commenting out the code or using `#if 0`, as disabled tests are still compiled (and thus won't rot).

To include disabled tests in test execution, just invoke the test program with the `--gtest_also_run_disabled_tests` flag.

Is it OK if I have two separate `TEST(Foo, Bar)` test methods defined in different namespaces?

Yes.

The rule is **all test methods in the same test suite must use the same fixture class.** This means that the following is **allowed** because both tests use the same fixture class (`::testing::Test`).

```
namespace foo {
TEST(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace foo

namespace bar {
TEST(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace bar
```

However, the following code is **not allowed** and will produce a runtime error from GoogleTest because the test methods are using different test fixture classes with the same test suite name.

```
namespace foo {
class CoolTest : public ::testing::Test {};  // Fixture foo::CoolTest
TEST_F(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace foo

namespace bar {
class CoolTest : public ::testing::Test {};  // Fixture: bar::CoolTest
TEST_F(CoolTest, DoSomething) {
  SUCCEED();
}
}  // namespace bar
```