If you cannot find the answer to your question here, and you have read Primer and AdvancedGuide, send it to googletestframework@googlegroups.com.

Why should I use Google Test instead of my favorite C++ testing framework?

First, let us say clearly that we don't want to get into the debate of which C++ testing framework is **the best**. There exist many fine frameworks for writing C++ tests, and we have tremendous respect for the developers and users of them. We don't think there is (or will be) a single best framework - you have to pick the right tool for the particular task you are tackling.

We created Google Test because we couldn't find the right combination of features and conveniences in an existing framework to satisfy *our* needs. The following is a list of things that *we* like about Google Test. We don't claim them to be unique to Google Test - rather, the combination of them makes Google Test the choice for us. We hope this list can help you decide whether it is for you too.

- Google Test is designed to be portable: it doesn't require exceptions or RTTI; it works around various bugs in various compilers and environments; etc. As a result, it works on Linux, Mac OS X, Windows and several embedded operating systems.
- Nonfatal assertions (`EXPECT_*`) have proven to be great time savers, as they allow a test to report multiple failures in a single edit-compile-test cycle.
- It's easy to write assertions that generate informative messages: you just use the stream syntax to append any additional information, e.g. `ASSERT_EQ(5, Foo(i)) << " where i = " << i;` . It doesn't require a new set of macros or special functions.
- Google Test automatically detects your tests and doesn't require you to enumerate them in order to run them.
- Death tests are pretty handy for ensuring that your asserts in production code are triggered by the right conditions.
- `SCOPED_TRACE` helps you understand the context of an assertion failure when it comes from inside a sub-routine or loop.
- You can decide which tests to run using name patterns. This saves time when you want to quickly reproduce a test failure.
- Google Test can generate XML test result reports that can be parsed by popular continuous build system like Hudson.
- Simple things are easy in Google Test, while hard things are possible: in addition to advanced features like global test environments and tests parameterized by values or types, Google Test supports various ways for the user to extend the framework -- if Google Test doesn't do something out of the box, chances are that a user can implement the feature using Google Test's public API, without changing Google Test itself. In particular, you can:
  - expand your testing vocabulary by defining custom predicates,
  - teach Google Test how to print your types,
  - define your own testing macros or utilities and verify them using Google Test's Service Provider Interface, and
  - reflect on the test cases or change the test output format by intercepting the test events.

I'm getting warnings when compiling Google Test. Would you fix them?

We strive to minimize compiler warnings Google Test generates. Before releasing a new version, we test to make sure that it doesn't generate warnings when compiled using its CMake script on Windows, Linux, and Mac OS.

Unfortunately, this doesn't mean you are guaranteed to see no warnings when compiling Google Test in your environment:

- You may be using a different compiler as we use, or a different version of the same compiler. We cannot possibly test for all compilers.
- You may be compiling on a different platform as we do.
- Your project may be using different compiler flags as we do.

It is not always possible to make Google Test warning-free for everyone. Or, it may not be desirable if the warning is rarely enabled and fixing the violations makes the code more complex.

If you see warnings when compiling Google Test, we suggest that you use the `-isystem` flag (assuming your are using GCC) to mark Google Test headers as system headers. That'll suppress warnings from Google Test headers.

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

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 containers 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(TestCaseName, TestName)` generates a class named `TestCaseName_TestName_Test` . What happens if `TestCaseName` or `TestName` contains `_` ?

1. If `TestCaseName` 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 `TestCaseName` ends with an `_` (say, `Foo_`), we get `Foo__TestName_Test` , which is invalid.
3. If `TestName` starts with an `_` (say, `_Bar`), we get `TestCaseName__Bar_Test` , which is invalid.
4. If `TestName` ends with an `_` (say, `Bar_`), we get `TestCaseName_Bar__Test` , which is invalid.

So clearly `TestCaseName` and `TestName` cannot start or end with `_` (Actually, `TestCaseName` 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 `TestCaseName` 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_Files_Like_An_Arrow_Test`). That's not good.

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

If you violate the rule, there may not be immediately 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 Google Test. Therefore it's best to follow the rule.

Why is it not recommended to install a pre-compiled copy of Google Test (for example, into /usr/local)?

In the early days, we said that you could install compiled Google Test libraries on `*nix` systems using `make install`. Then every user of your machine can write tests without recompiling Google Test.

This seemed like a good idea, but it has a got-cha: every user needs to compile his tests using the *same* compiler flags used to compile the installed Google Test libraries; otherwise he may run into undefined behaviors (i.e. the tests can behave strangely and may even crash for no obvious reasons).

Why? Because C++ has this thing called the One-Definition Rule: if two C++ source files contain different definitions of the same class/function/variable, and you link them together, you violate the rule. The linker may or may not catch the error (in many cases it's not required by the C++ standard to catch the violation). If it doesn't, you get strange run-time behaviors that are unexpected and hard to debug.

If you compile Google Test and your test code using different compiler flags, they may see different definitions of the same class/function/variable (e.g. due to the use of `#if` in Google Test). Therefore, for your sanity, we recommend to avoid installing pre-compiled Google Test libraries. Instead, each project should compile Google Test itself such that it can be sure that the same flags are used for both Google Test and the tests.

How do I generate 64-bit binaries on Windows (using Visual Studio 2008)?

(Answered by Trevor Robinson)

Load the supplied Visual Studio solution file, either `msvc\gtest-md.sln` or `msvc\gtest.sln`. Go through the migration wizard to migrate the solution and project files to Visual Studio 2008. Select `Configuration Manager...` from the `Build` menu. Select `<New...>` from the `Active solution platform` dropdown. Select `x64` from the new platform dropdown, leave `Copy settings from` set to `Win32` and `Create new project platforms` checked, then click `OK`. You now have `Win32` and `x64` platform configurations, selectable from the `Standard` toolbar, which allow you to toggle between building 32-bit or 64-bit binaries (or both at once using Batch Build).

In order to prevent build output files from overwriting one another, you'll need to change the `Intermediate Directory` settings for the newly created platform configuration across all the projects. To do this, multi-select (e.g. using shift-click) all projects (but not the solution) in the `Solution Explorer`. Right-click one of them and select `Properties`. In the left pane, select `Configuration Properties`, and from the `Configuration` dropdown, select `All Configurations`. Make sure the selected platform is `x64`. For the `Intermediate Directory` setting, change the value from `$(PlatformName)\$(ConfigurationName)` to `$(OutDir)\$(ProjectName)`. Click `OK` and then build the solution. When the build is complete, the 64-bit binaries will be in the `msvc\x64\Debug` directory.

Can I use Google Test on MinGW?

We haven't tested this ourselves, but Per Abrahamsen reported that he was able to compile and install Google Test successfully when using MinGW from Cygwin. You'll need to configure it with:

```
PATH/TO/configure CC="gcc -mno-cygwin" CXX="g++ -mno-cygwin"
```

You should be able to replace the `-mno-cygwin` option with direct links to the real MinGW binaries, but we haven't tried that.

Caveats:

- There are many warnings when compiling.
- `make check` will produce some errors as not all tests for Google Test itself are compatible with MinGW.

We also have reports on successful cross compilation of Google Test MinGW binaries on Linux using these instructions on the WxWidgets site.

Please contact `googletestframework@googlegroups.com` if you are interested in improving the support for MinGW.

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

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 Google Test harder to maintain and more error-prone than necessary).

The `EXPECT_EQ()` macro takes the *expected* value as its first argument and the *actual* value as the second. It's reasonable that someone wants 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)` isn'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'll have to support `EXPECT_NE(ptr, NULL)` as well, as unlike `EXPECT_EQ`, we don't have a convention on the order of the two arguments for `EXPECT_NE`. 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 Google Mock's 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.

Does Google Test support running tests in parallel?

Test runners tend to be tightly coupled with the build/test environment, and Google Test doesn't try to solve the problem of running tests in parallel. Instead, we tried to make Google Test work nicely with test runners. For example, Google Test's XML report contains the time spent on each test, and its `gtest_list_tests` and `gtest_filter` flags can be used for splitting the execution of test methods into multiple processes. These functionalities can help the test runner run the tests in parallel.

Why don't Google Test run the tests in different threads to speed things up?

It's difficult to write thread-safe code. Most tests are not written with thread-safety in mind, and thus may not work correctly in a multi-threaded setting.

If you think about it, it's already hard to make your code work when you know what other threads are doing. It's much harder, and sometimes even impossible, to make your code work when you don't know what other threads are doing (remember that test methods can be added, deleted, or modified after your test was written). If you want to run the tests in parallel, you'd better run them in different processes.

Why aren't Google Test assertions implemented using exceptions?

Our original motivation was to be able to use Google Test in projects that disable exceptions. Later we realized some additional benefits of this approach:

1. Throwing in a destructor is undefined behavior in C++. Not using exceptions means Google Test's assertions are safe to use in destructors.
2. The `EXPECT_*` family of macros will continue even after a failure, allowing multiple failures in a `TEST` to be reported in a single run. This is a popular feature, as in C++ the edit-compile-test cycle is usually quite long and being able to fixing more than one thing at a time is a blessing.
3. If assertions are implemented using exceptions, a test may falsely ignore a failure if it's caught by user code:

   ```
   try { ... ASSERT_TRUE(...) ... }
   catch (...) { ... }
   ```

   The above code will pass even if the `ASSERT_TRUE` throws. While it's unlikely for someone to write this in a test, it's possible to run into this pattern when you write assertions in callbacks that are called by the code under test.

The downside of not using exceptions is that `ASSERT_*` (implemented using `return`) will only abort the current function, not the current `TEST`.

Why do we use two different macros for tests with and without fixtures?

Unfortunately, C++'s macro system doesn't allow us to use the same macro for both cases. One possibility is to provide only one macro for tests with fixtures, and require the user to define an empty fixture sometimes:

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

TEST_F(FooTest, DoesThis) { ... }
```

or

```
typedef ::testing::Test FooTest;

TEST_F(FooTest, DoesThat) { ... }
```

Yet, many people think this is one line too many. :-) Our goal was to make it really easy to write tests, so we tried to make simple tests trivial to create. That means using a separate macro for such tests.

We think neither approach is ideal, yet either of them is reasonable. In the end, it probably doesn't matter much either way.

Why don't we use structs as test fixtures?

We like to use structs only when representing passive data. This distinction between structs and classes is good for documenting the intent of the code's author. Since test fixtures have logic like `SetUp()` and `TearDown()`, they are better defined as classes.

Why are death tests implemented as assertions instead of using a test runner?

Our goal was to make death tests as convenient for a user as C++ possibly allows. In particular:

- The runner-style requires to split the information into two pieces: the definition of the death test itself, and the specification for the runner on how to run the death test and what to expect. The death test would be written in C++, while the runner spec may or may not be. A user needs to carefully keep the two in sync. `ASSERT_DEATH(statement, expected_message)` specifies all necessary information in one place, in one language, without boilerplate code. It is very declarative.
- `ASSERT_DEATH` has a similar syntax and error-reporting semantics as other Google Test assertions, and thus is easy to learn.
- `ASSERT_DEATH` can be mixed with other assertions and other logic at your will. You are not limited to one death test per test method. For example, you can write something like:

  ```
  if (FooCondition()) {
    ASSERT_DEATH(Bar(), "blah");
  } else {
    ASSERT_EQ(5, Bar());
  }
  ```

  If you prefer one death test per test method, you can write your tests in that style too, but we don't want to impose that on the users. The fewer artificial limitations the better.
- `ASSERT_DEATH` can reference local variables in the current function, and you can decide how many death tests you want based on run-time information. For example,

  ```
  const int count = GetCount();  // Only known at run time.
  for (int i = 1; i <= count; i++) {
    ASSERT_DEATH({
      double* buffer = new double[i];
      ... initializes buffer ...
      Foo(buffer, i)
    }, "blah blah");
  }
  ```

  The runner-based approach tends to be more static and less flexible, or requires more user effort to get this kind of flexibility.

Another interesting thing about `ASSERT_DEATH` is that it calls `fork()` to create a child process to run the death test. This is lightening fast, as `fork()` uses copy-on-write pages and incurs almost zero overhead, and the child process starts from the user-supplied statement directly, skipping all global and local initialization and any code leading to the given statement. If you launch the child process from scratch, it can take seconds just to load everything and start running if the test links to many libraries dynamically.

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.

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 Google Test comparison assertions (`EXPECT_EQ` , etc) will generate an "undefined reference" linker error.

I have an interface that has several implementations. Can I write a set of tests once and repeat them over all the implementations?

Google Test doesn't yet have good support for this kind of tests, or data-driven tests in general. We hope to be able to make improvements in this area soon.

Can I derive a test fixture from another?

Yes.

Each test fixture has a corresponding and same named test case. This means only one test case 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 cases don't leak important system resources like fonts and brushes.

In Google Test, you share a fixture among test cases by putting the shared logic in a base test fixture, then deriving from that base a separate fixture for each test case 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:
    virtual void SetUp() {
      BaseTest::SetUp();  // Sets up the base fixture first.
      ... additional set-up work ...
    }
    virtual void TearDown() {
      ... 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. Google Test has no limit on how deep the hierarchy can be.

For a complete example using derived test fixtures, see sample5.

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.

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

In Google Test, 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. Please make sure you have read this.

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()`.

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 is no race conditions or dead locks in your program. No silver bullet - sorry!

Should I use the constructor/destructor of the test fixture or the set-up/tear-down function?

The first thing to remember is that Google Test does not reuse the same test fixture object across multiple tests. For each `TEST_F`, Google Test will create a fresh test fixture object, *immediately* call `SetUp()`, run the test, call `TearDown()`, and then *immediately* delete the test fixture object. Therefore, there is no need to write a `SetUp()` or `TearDown()` function if the constructor or destructor already does the job.

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

- 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 Google Test 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 Google Test assertions in a destructor if your code could run on such a platform.
- In a constructor or destructor, you cannot make a virtual function call on this object. (You can call a method declared as virtual, but it will be statically bound.) Therefore, if you need to call a method that will be overriden in a derived class, you have to use `SetUp()/TearDown()`.

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

If the predicate function you use in `ASSERT_PRED*` or `EXPECT_PRED*` is overloaded or a template, the compiler will have trouble figuring out which overloaded version it should use. `ASSERT_PRED_FORMAT*` and `EXPECT_PRED_FORMAT*` don't have this problem.

If you see this error, you might want to switch to `(ASSERT|EXPECT)_PRED_FORMAT*`, which will also give you a better failure message. If, however, that is not an option, you can resolve the problem by explicitly telling the compiler which version to pick.

For example, suppose you have

```
bool IsPositive(int n) {
  return n > 0;
}
bool IsPositive(double x) {
  return x > 0;
}
```

you will get a compiler error if you write

```
EXPECT_PRED1(IsPositive, 5);
```

However, this will work:

```
EXPECT_PRED1(*static_cast<bool (*)(int)>*(IsPositive), 5);
```

(The stuff inside the angled brackets for the `static_cast` operator is the type of the function pointer for the `int`-version of `IsPositive()`.)

As another example, when you have a template function

```
template <typename T>
bool IsNegative(T x) {
  return x < 0;
}
```

you can use it in a predicate assertion like this:

```
ASSERT_PRED1(IsNegative*<int>*, -5);
```

Things are more interesting if your template has more than one parameters. The following won't compile:

```
ASSERT_PRED2(*GreaterThan<int, int>*, 5, 0);
```

as the C++ pre-processor thinks you are giving `ASSERT_PRED2` 4 arguments, which is one more than expected. The workaround is to wrap the predicate function in parentheses:

```
ASSERT_PRED2(*(GreaterThan<int, int>)*, 5, 0);
```

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. A test runner 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 Google Test assertion failure. Very bad.

To help the users avoid this dangerous bug, the implementation of `RUN_ALL_TESTS()` causes gcc to raise this warning, when the return value is ignored. If you see this warning, the fix is simple: just make sure its value is used as the return value of `main()`.

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 set-up function is not called. Why?

C++ is case-sensitive. It should be spelled as `SetUp()`. Did you spell it as `Setup()`?

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

How do I jump to the line of a failure in Emacs directly?

Google Test's failure message format is understood by Emacs and many other IDEs, like acme and XCode. If a Google Test message is in a compilation buffer in Emacs, then it's clickable. You can now hit `enter` on a message

to jump to the corresponding source code, or use `C-x `` to jump to the next failure.

I have several test cases 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) { ... }
```

The Google Test output is buried in a whole bunch of log messages. What do I do?

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

Since most log messages go to stderr, we decided to let Google Test output go to stdout. This way, you can easily separate the two using redirection. For example:

```
./my_test > googletest_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 cases have something in common.

How do I test private class members without writing FRIEND_TEST()s?

You should try to write testable code, which means classes should be easily tested from their public interface. One way to achieve this is the Pimpl idiom: you move all private members of a class into a helper class, and make all members of the helper class public.

You have several other options that don't require using `FRIEND_TEST`:

- Write the tests as members of the fixture class:

```
class Foo {
friend class FooTest;
...
};
```

class FooTest : public ::testing::Test { protected: ... void Test1() {...} // This accesses private members of class Foo. void Test2() {...} // So does this one. };

TEST_F(FooTest, Test1) { Test1(); }

TEST_F(FooTest, Test2) { Test2(); }

```
  * In the fixture class, write accessors for the tested class' private members, then
use the accessors in your tests:
```

class Foo { friend class FooTest; ... };

class FooTest : public ::testing::Test { protected: ... T1 get_private_member1(Foo* obj) { return obj->private_member1_; } };

TEST_F(FooTest, Test1) { ... get_private_member1(x) ... }

```
  * If the methods are declared **protected**, you can change their access level in a
test-only subclass:
```

class YourClass { ... protected: // protected access for testability. int DoSomethingReturningInt(); ... };

// in the your_class_test.cc file: class TestableYourClass : public YourClass { ... public: using YourClass::DoSomethingReturningInt; // changes access rights ... };

TEST_F(YourClassTest, DoSomethingTest) { TestableYourClass obj; assertEquals(expected_value, obj.DoSomethingReturningInt()); }

```
## How do I test private class static members without writing FRIEND\_TEST()s? ##

We find private static methods clutter the header file.  They are
implementation details and ideally should be kept out of a .h. So often I make
them free functions instead.

Instead of:
```

// foo.h class Foo { ... private: static bool Func(int n); };

// foo.cc bool Foo::Func(int n) { ... }

// foo_test.cc EXPECT_TRUE(Foo::Func(12345));

```
You probably should better write:
```

// foo.h class Foo { ... };

// foo.cc namespace internal { bool Func(int n) { ... } }

// foo_test.cc namespace internal { bool Func(int n); }

EXPECT_TRUE(internal::Func(12345));

```
## I would like to run a test several times with different parameters. Do I need to
write several similar copies of it? ##

No. You can use a feature called [value-parameterized tests]
(V1_6_AdvancedGuide.md#Value_Parameterized_Tests) which
lets you repeat your tests with different parameters, without defining it more than
once.

## How do I test a file that defines main()? ##

To test a `foo.cc` file, you need to compile and link it into your unit test
program. However, when the file contains a definition for the `main()`
function, it will clash with the `main()` of your unit test, and will result in
a build error.

The right solution is to split it into three files:
  1. `foo.h` which contains the declarations,
  1. `foo.cc` which contains the definitions except `main()`, and
  1. `foo_main.cc` which contains nothing but the definition of `main()`.

Then `foo.cc` can be easily tested.

If you are adding tests to an existing file and don't want an intrusive change
like this, there is a hack: just include the entire `foo.cc` file in your unit
test. For example:
```

// File foo_unittest.cc

// The headers section ...

// Renames main() in foo.cc to make room for the unit test main() #define main FooMain

#include "a/b/foo.cc"

// The tests start here. ...

```


However, please remember this is a hack and should only be used as the last
resort.
```

```
## What can the statement argument in ASSERT\_DEATH() be? ##

`ASSERT_DEATH(_statement_, _regex_)` (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 any where 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");}

```
`googletest_unittest.cc` contains more examples if you are interested.

## What syntax does the regular expression in ASSERT\_DEATH use? ##

On POSIX systems, Google Test uses the POSIX Extended regular
expression syntax
(http://en.wikipedia.org/wiki/Regular_expression#POSIX_Extended_Regular_Expressions).
On Windows, it uses a limited variant of regular expression
syntax. For more details, see the
[regular expression syntax](V1_6_AdvancedGuide.md#Regular_Expression_Syntax).

## I have a fixture class Foo, but TEST\_F(Foo, Bar) gives me error "no matching
function for call to Foo::Foo()". Why? ##

Google Test 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 `Foo`, then you need
to define a default constructor, even if it would be empty.
  * If `Foo` 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 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 Google Test require the entire test case, instead of individual tests, to
be named FOODeathTest when it uses ASSERT\_DEATH? ##

Google Test does not interleave tests from different test cases. That is, it
runs all tests in one test case first, and then runs all tests in the next test
case, and so on. Google Test does this because it needs to set up a test case
before the first test in it is run, and tear it down afterwords. 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 cases, 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 case FOODeathTest 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 case 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) { ... }

typedef FooTest FooDeathTest;

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

```
## 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.

## How do I suppress the memory leak messages on Windows? ##

Since the statically initialized Google Test 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.

## I am building my project with Google Test in Visual Studio and all I'm getting is a
bunch of linker errors (or warnings). Help! ##

You may get a number of the following linker error or warnings if you
attempt to link your test project with the Google Test library when
your project and the are not built using the same compiler settings.

  * LNK2005: symbol already defined in object
  * LNK4217: locally defined symbol 'symbol' imported in function 'function'
  * LNK4049: locally defined symbol 'symbol' imported

The Google Test project (gtest.vcproj) has the Runtime Library option
set to /MT (use multi-threaded static libraries, /MTd for debug). If
your project uses something else, for example /MD (use multi-threaded
DLLs, /MDd for debug), you need to change the setting in the Google
Test project to match your project's.

To update this setting open the project properties in the Visual
Studio IDE then select the branch Configuration Properties | C/C++ |
Code Generation and change the option "Runtime Library".  You may also try
using gtest-md.vcproj instead of gtest.vcproj.

## I put my tests in a library and Google Test doesn't run them. What's happening? ##
Have you read a
[warning](V1_6_Primer.md#important-note-for-visual-c-users) on
the Google Test Primer page?

## I want to use Google Test with Visual Studio but don't know where to start. ##
Many people are in your position and one of the posted his solution to
our mailing list. Here is his link:
http://hassanjamilahmad.blogspot.com/2009/07/gtest-starters-help.html.

## I am seeing compile errors mentioning std::type\_traits when I try to use Google
Test on Solaris. ##
Google Test uses parts of the standard C++ library that SunStudio does not support.
Our users reported success using alternative implementations. Try running the build
```

```
after runing this commad:

`export CC=cc CXX=CC CXXFLAGS='-library=stlport4'`

## 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](http://en.wikipedia.org/wiki/Unusual_software_bug#Heisenbug).
Therefore we strongly advise against the practice, and Google Test 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](http://jamesshore.com/Blog/Dependency-
Injection-Demystified.html).
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, 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.

## Google Test defines a macro that clashes with one defined by another library. How do
I deal with that? ##

In C++, macros don't obey namespaces.  Therefore two libraries that
both define a macro of the same name will clash if you `#include` both
definitions.  In case a Google Test macro clashes with another
library, you can force Google Test to rename its macro to avoid the
conflict.

Specifically, if both Google Test and some other code define macro
`FOO`, you can add
```

-DGTEST_DONT_DEFINE_FOO=1

```
to the compiler flags to tell Google Test to change the macro's name
from `FOO` to `GTEST_FOO`. For example, with `-DGTEST_DONT_DEFINE_TEST=1`, you'll need
to write
```

GTEST_TEST(SomeTest, DoesThis) { ... }

```
instead of
```

TEST(SomeTest, DoesThis) { ... }

```
in order to define a test.
```

```
Currently, the following `TEST`, `FAIL`, `SUCCEED`, and the basic comparison assertion
macros can have alternative names. You can see the full list of covered macros [here]
(http://www.google.com/codesearch?
q=if+!GTEST_DONT_DEFINE_\w%2B+package:http://googletest\.googlecode\.com+file:/include/gt
 More information can be found in the "Avoiding Macro Name Clashes" section of the
README file.

## My question is not covered in your FAQ! ##

If you cannot find the answer to your question in this FAQ, there are
some other resources you can use:

  1. read other [wiki pages](http://code.google.com/p/googletest/w/list),
  1. search the mailing list [archive]
(http://groups.google.com/group/googletestframework/topics),
  1. ask it on [googletestframework@googlegroups.com]
(mailto:googletestframework@googlegroups.com) and someone will answer it (to prevent
spam, we require you to join the [discussion group]
(http://groups.google.com/group/googletestframework) before you can post.).

Please note that creating an issue in the
[issue tracker](http://code.google.com/p/googletest/issues/list) is _not_
a good way to get your answer, as it is monitored infrequently by a
very small number of people.

When asking a question, it's helpful to provide as much of the
following information as possible (people cannot help you if there's
not enough information in your question):

  * the version (or the revision number if you check out from SVN directly) of Google
Test you use (Google Test is under active development, so it's possible that your
problem has been solved in a later version),
  * your operating system,
  * the name and version of your compiler,
  * the complete command line flags you give to your compiler,
  * the complete compiler error messages (if the question is about compilation),
  * the _actual_ code (ideally, a minimal but complete program) that has the problem
you encounter.
```