Google Test

TODO: Advanced googletest Topics

Making Good Test

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

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

Basic Concepts

When using googletest, you start by writing assertions. Assertions are statements that check whether a condition is true. An assertion's result can be *success*, *nonfatal failure*, or *fatal failure*.

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

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

A *test program* can contain multiple test suites.

Assertions

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

The assertions come in pairs that test the same thing but have different effects on the current function. `ASSERT_*` versions generate a fatal failure when they fail, **abort the current function**. `EXPECT_*` versions generate nonfatal failures, which don't about the current function.

Since `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.

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.

Basic Assertions

Fatal Assertion	Nonfatal Assertion	Verified
`ASSERT_TRUE(condition);`	`EXPECT_TRUE(condition);`	`condition` is true
`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	`condition` is false

Binary Comparisons

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`

When user-defined types, it may be necessary to use `ASSERT_TRUE()`, or `EXPECT_TRUE()` to assert the equality of two objects of a user-defined type.

However, when possible, `ASSERT_EQ(ACUTAL, EXPECTED)` is proffered to `ASSERT_TRUE(ACTUAL == EXPECTED)`, since it tells you `actual` and `expected`'s values on failure.

`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 by value, use `ASSERT_STREQ()` (which is described later on). In particular, to assert that a C string is `NULL`, use `ASSERT_STREQ(c_string, NULL)`. Consider using `ASSER_EQ(c_string, nullptr)`.

When doing pointer comparisons use `*_EQ(ptr, nullptr)` and `*_NE(ptr, nullptr)` instead of `*_EQ(ptr, NULL)`, and `*_NE(ptr, NULL)`. This is because `nullptr` is typed, while `NULL` is not.

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	Verifies
`ASSERT_STREQ(str1,str2);`	`EXPECT_STREQ(str1,str2);`	the two C strings have the same content
`ASSERT_STRNE(str1,str2);`	`EXPECT_STRNE(str1,str2);`	the two C strings have different contents
`ASSERT_STRCASEEQ(str1,str2);`	`EXPECT_STRCASEEQ(str1,str2);`	the two C strings have the same content, ignoring case
`ASSERT_STRCASENE(str1,str2);`	`EXPECT_STRCASENE(str1,str2);`	the two C strings have different contents, ignoring case