

Google Test Quick Reference

Example Tests

- The following code snippet is a simple test that one might write when driving a bowling scorer object.

```
TEST(BowlingGameTest, AllGuttersGetsZero) {  
    BowlingGame game;  
    vector<int> rolls(20, 0);  
    ASSERT_EQ(0, game.Score(rolls));  
}
```

- The following code snippets demonstrate a simple data fixture and a test that uses that fixture.

```
class QueueTest : public ::testing::Test {  
protected:  
    virtual void SetUp() {  
        q1_.Enqueue(1);  
    }  
  
    Queue<int> q0_;  
    Queue<int> q1_;  
};
```

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

```
TEST_F(QueueTest, DequeueWorks) {  
    int* n = q1_.Dequeue();  
    ASSERT_TRUE(n != NULL);  
    EXPECT_EQ(1, *n);  
    EXPECT_EQ(0, q1_.size());  
    delete n;  
}
```

Creating Tests

TEST(test_case_name, test_name) test_case_name is the name of the Test Case. test_name is the name of the individual scenario contained within this test method. A test case can and probably should contain multiple tests.

TEST_F(test_case_name, test_name) This is the same as above except the test_case_name is the name of a *test fixture*. *Test fixtures* allow you to use the same data and setup across multiple tests.

* ASSERT_* yields a fatal failure and returns from the current function, while EXPECT_* yields a nonfatal failure, allowing the function to continue running.

Basic Assertion

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_TRUE(<i>condition</i>)	EXPECT_TRUE(<i>condition</i>)	<i>condition</i> is true
ASSERT_FALSE(<i>condition</i>)	EXPECT_FALSE(<i>condition</i>)	<i>condition</i> is false

Explicit Success of Failure

Assertion	Description
SUCCEED()	This is purely documentation and currently doesn't generate any visible output. In the future google may choose to have this macro produce messages.
FAIL()	This will generate a fatal failure.
ADD_FAILURE()	This will generate a nonfatal failure.

Binary Comparison

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_EQ(<i>expected, actual</i>)	EXPECT_EQ(<i>expected, actual</i>)	<i>expected == actual</i>
ASSERT_NE(<i>val1, val2</i>)	EXPECT_NE(<i>val1, val2</i>)	<i>val1 != val2</i>
ASSERT_LT(<i>val1, val2</i>)	EXPECT_LT(<i>val1, val2</i>)	<i>val1 < val2</i>
ASSERT_LE(<i>val1, val2</i>)	EXPECT_LE(<i>val1, val2</i>)	<i>val1 <= val2</i>
ASSERT_GT(<i>val1, val2</i>)	EXPECT_GT(<i>val1, val2</i>)	<i>val1 > val2</i>
ASSERT_GE(<i>val1, val2</i>)	EXPECT_GE(<i>val1, val2</i>)	<i>val1 >= val2</i>

String Comparison

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_STREQ(<i>expected, actual</i>)	EXPECT_STREQ(<i>expected, actual</i>)	the two C strings have the same content
ASSERT_STRNE(<i>str1, str2</i>)	EXPECT_STRNE(<i>str1, str2</i>)	the two C strings have different content
ASSERT_STRCASEEQ(<i>exp, act</i>)	EXPECT_STRCASEEQ(<i>exp, act</i>)	the two C strings have the same content, ignoring case
ASSERT_STRCASENE(<i>str1, str2</i>)	EXPECT_STRCASENE(<i>str1, str2</i>)	the two C strings have different content, ignoring case.

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

* CASE in an assertion name means that case is ignored.

Exception Assertions

Fatal Assertion	Nonfatal Assertion	Verifies
<code>ASSERT_THROW(stmt, exc_type)</code>	<code>EXPECT_THROW(stmt, exc_type)</code>	<i>stmt</i> throws an exception of the given <i>exc_type</i> .
<code>ASSERT_ANY_THROW(stmt)</code>	<code>EXPECT_ANY_THROW(stmt)</code>	<i>stmt</i> throws an exception of any type.
<code>ASSERT_NO_THROW(stmt)</code>	<code>EXPECT_NO_THROW(stmt)</code>	<i>stmt</i> throws no exception

Examples:

```
ASSERT_THROW(Foo(5), bar_exception);
```

```
EXPECT_NO_THROW({
    int n = 5;
    Bar(&n)
});
```

Floating Point Comparison

Fatal Assertion	Nonfatal Assertion	Verifies
<code>ASSERT_FLOAT_EQ(exp, act)</code>	<code>EXPECT_FLOAT_EQ(exp, act)</code>	The two float values are almost equal.
<code>ASSERT_DOUBLE_EQ(exp, act)</code>	<code>EXPECT_DOUBLE_EQ(exp, act)</code>	The two double values are almost equal
<code>ASSERT_NEAR(val1, val2, abs_err)</code>	<code>EXPECT_NEAR(val1, val2, abs_err)</code>	The difference between the <i>val1</i> and <i>val2</i> doesn't exceed <i>abs_err</i> .

* By "almost equal" we mean the two values are within 4 ULP's from each other.

Windows HRESULT Assertions

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_HRESULT_SUCCEEDED(<i>exp</i>)	EXPECT_HRESULT_SUCCEEDED(<i>exp</i>)	<i>exp</i> is a success HRESULT
ASSERT_HRESULT_FAILED(<i>exp</i>)	EXPECT_HRESULT_FAILED(<i>exp</i>)	<i>exp</i> is a success HRESULT

* The generated output contains the human-readable error message associated with the HRESULT code returned by *exp*.

Predicate Assertions for Better Error Messages

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_PRED1(<i>pred</i> , <i>val1</i>)	EXPECT_PRED1(<i>pred</i> , <i>val1</i>)	<i>pred(val1)</i> returns true
ASSERT_PRED2(<i>pred</i> , <i>val1</i> , <i>val2</i>)	EXPECT_PRED2(<i>pred</i> , <i>val1</i> , <i>val2</i>)	<i>pred(val1, val2)</i> returns true
...

* When the assertion fails it prints the value for each argument.

* Currently we only provide predicate assertions of arity ≤ 5 .

Example:

```
bool MutuallyPrime(int m, int n) {...}
const int a = 3;
const int b = 4;
const int c = 10;
```

The assertion `EXPECT_PRED2(MutuallyPrime, a, b)` will succeed, while the assertion `EXPECT_PRED2(MutuallyPrime(b, c))` will fail with the message:

```
MutuallyPrime(b, c) is false, where
b is 4
c is 10
```

Running Test Programs - Advanced Options

Running a Subset of the Tests

If you set the *GTEST_FILTER* environment variable or the *--gtest_filter* flag to a filter string Google Test will only run those tests whose full name match the filter.

Temporarily Disabling Tests

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.

* Note: This feature should only be used for temporary pain-relief. You still have to fix the disabled tests at a later time. As a reminder, Google Test will print a banner warning you if a test program contains disabled tests.

Temporarily Enabling Disabled Tests

To include disabled tests in test execution, just invoke the test program with *--gtest_also_run_disabled_tests* or set the *GTEST_ALSO_RUN_DISABLED_TESTS* environment variable to a value greater than 0.

Colored Terminal Output

You can set the *GTEST_COLOR* environment variable or set the *--gtest_color* command line flag to *yes*, *no*, or *auto* to enable colors, disable colors or let Google Test decide.

Suppressing Elapsed Time

By default, Google Test prints the time it takes to run each test. To suppress that, run the program with the *--gtest_print_time=0* command line flag. Setting the *GTEST_PRINT_TIME* environment variable to 0 has the same effect.

Generating an XML Report

Google Test can emit a detailed XML report to a file in addition to its normal textual output. To generate the XML report, set the *GTEST_OUTPUT* environment variable or the *--gtest_output* flag to the string "*xml:path_to_output_file*", which will create the file at the given location. You can also just use the string "*xml*", in which case the output can be found in the *test_detail.xml* file in the current directory. If the file already exists it will pick a different name (append number) to avoid overwriting it.

The report uses a format based on the *junitreport* Ant task and can be parsed by popular Continuous Integration servers like Hudson.