Assertions Reference

This page lists the assertion macros provided by GoogleTest for verifying code behavior. To use them, add `#include <gtest/gtest.h>` .

The majority of the macros listed below come as a pair with an `EXPECT_` variant and an `ASSERT_` variant. Upon failure, `EXPECT_` macros generate nonfatal failures and allow the current function to continue running, while `ASSERT_` macros generate fatal failures and abort the current function.

All assertion macros support streaming a custom failure message into them with the `<<` operator, for example:

```
EXPECT_TRUE(my_condition) << "My condition is not true";
```

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.

Explicit Success and Failure

The assertions in this section generate a success or failure directly instead of testing a value or expression. These are useful when control flow, rather than a Boolean expression, determines the test's success or failure, as shown by the following example:

```
switch(expression) {
  case 1:
    ... some checks ...
  case 2:
    ... some other checks ...
  default:
    FAIL() << "We shouldn't get here.";
}
```

SUCCEED

`SUCCEED()`

Generates a success. This *does not* make the overall test succeed. A test is considered successful only if none of its assertions fail during its execution.

The `SUCCEED` assertion is purely documentary and currently doesn't generate any user-visible output. However, we may add `SUCCEED` messages to GoogleTest output in the future.

FAIL

`FAIL()`

Generates a fatal failure, which returns from the current function.

Can only be used in functions that return `void` . See Assertion Placement for more information.

ADD_FAILURE

`ADD_FAILURE()`

Generates a nonfatal failure, which allows the current function to continue running.

ADD_FAILURE_AT

`ADD_FAILURE_AT( file_path , line_number )`

Generates a nonfatal failure at the file and line number specified.

Generalized Assertion

The following assertion allows matchers to be used to verify values.

EXPECT_THAT

`EXPECT_THAT( value , matcher )`
`ASSERT_THAT( value , matcher )`

Verifies that `value` matches the matcher `matcher`.

For example, the following code verifies that the string `value1` starts with `"Hello"`, `value2` matches a regular expression, and `value3` is between 5 and 10:

```
#include <gmock/gmock.h>

using ::testing::AllOf;
using ::testing::Gt;
using ::testing::Lt;
using ::testing::MatchesRegex;
using ::testing::StartsWith;

...
EXPECT_THAT(value1, StartsWith("Hello"));
EXPECT_THAT(value2, MatchesRegex("Line \\d+"));
ASSERT_THAT(value3, AllOf(Gt(5), Lt(10)));
```

Matchers enable assertions of this form to read like English and generate informative failure messages. For example, if the above assertion on `value1` fails, the resulting message will be similar to the following:

```
Value of: value1
  Actual: "Hi, world!"
Expected: starts with "Hello"
```

GoogleTest provides a built-in library of matchers—see the Matchers Reference. It is also possible to write your own matchers—see Writing New Matchers Quickly. The use of matchers makes `EXPECT_THAT` a powerful, extensible assertion.

The idea for this assertion was borrowed from Joe Walnes' Hamcrest project, which adds `assertThat()` to JUnit.

Boolean Conditions

The following assertions test Boolean conditions.

EXPECT_TRUE

`EXPECT_TRUE( condition )`
`ASSERT_TRUE( condition )`

Verifies that `condition` is true.

EXPECT_FALSE

`EXPECT_FALSE( condition )`
`ASSERT_FALSE( condition )`

Verifies that `condition` is false.

Binary Comparison

The following assertions compare two values. The value arguments must be comparable by the assertion's comparison operator, otherwise a compiler error will result.

If an argument supports the `<<` operator, it will be called to print the argument when the assertion fails. Otherwise, GoogleTest will attempt to print them in the best way it can—see Teaching GoogleTest How to Print Your Values.

Arguments are always evaluated exactly once, so it's OK for the arguments to have side effects. However, the argument evaluation order is undefined and programs should not depend on any particular argument evaluation order.

These assertions work with both narrow and wide string objects (`string` and `wstring`).

See also the Floating-Point Comparison assertions to compare floating-point numbers and avoid problems caused by rounding.

EXPECT_EQ

```
EXPECT_EQ( val1 , val2 )
ASSERT_EQ( val1 , val2 )
```

Verifies that *val1* `==` *val2* .

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. Use `EXPECT_STREQ` to compare C strings (e.g. `const char*`) by value.

When comparing a pointer to `NULL` , use `EXPECT_EQ( ptr , nullptr)` instead of `EXPECT_EQ( ptr , NULL)` .

EXPECT_NE

```
EXPECT_NE( val1 , val2 )
ASSERT_NE( val1 , val2 )
```

Verifies that *val1* `!=` *val2* .

Does pointer equality on pointers. If used on two C strings, it tests if they are in different memory locations, not if they have different values. Use `EXPECT_STRNE` to compare C strings (e.g. `const char*`) by value.

When comparing a pointer to `NULL` , use `EXPECT_NE( ptr , nullptr)` instead of `EXPECT_NE( ptr , NULL)` .

EXPECT_LT

```
EXPECT_LT( val1 , val2 )
ASSERT_LT( val1 , val2 )
```

Verifies that *val1* `<` *val2* .

EXPECT_LE

```
EXPECT_LE( val1 , val2 )
ASSERT_LE( val1 , val2 )
```

Verifies that *val1* `<=` *val2* .

EXPECT_GT

```
EXPECT_GT( val1 , val2 )
ASSERT_GT( val1 , val2 )
```

Verifies that *val1* `>` *val2* .

EXPECT_GE

```
EXPECT_GE( val1 , val2 )
ASSERT_GE( val1 , val2 )
```

Verifies that *val1* `>=` *val2* .

String Comparison

The following assertions compare two **C strings**. To compare two `string` objects, use `EXPECT_EQ` or `EXPECT_NE` instead.

These assertions 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.

To compare a C string with `NULL` , use `EXPECT_EQ( c_string , nullptr)` or `EXPECT_NE( c_string , nullptr)` .

EXPECT_STREQ

```
EXPECT_STREQ( str1 , str2 )
ASSERT_STREQ( str1 , str2 )
```

Verifies that the two C strings *`str1`* and *`str2`* have the same contents.

EXPECT_STRNE

```
EXPECT_STRNE( str1 , str2 )
ASSERT_STRNE( str1 , str2 )
```

Verifies that the two C strings *`str1`* and *`str2`* have different contents.

EXPECT_STRCASEEQ

```
EXPECT_STRCASEEQ( str1 , str2 )
ASSERT_STRCASEEQ( str1 , str2 )
```

Verifies that the two C strings *`str1`* and *`str2`* have the same contents, ignoring case.

EXPECT_STRCASENE

```
EXPECT_STRCASENE( str1 , str2 )
ASSERT_STRCASENE( str1 , str2 )
```

Verifies that the two C strings *`str1`* and *`str2`* have different contents, ignoring case.

Floating-Point Comparison

The following assertions compare two floating-point values.

Due to rounding errors, it is very unlikely that two floating-point values will match exactly, so `EXPECT_EQ` is not suitable. In general, for floating-point comparison to make sense, the user needs to carefully choose the error bound.

GoogleTest also provides assertions that use a default error bound based on Units in the Last Place (ULPs). To learn more about ULPs, see the article Comparing Floating Point Numbers.

EXPECT_FLOAT_EQ

```
EXPECT_FLOAT_EQ( val1 , val2 )
ASSERT_FLOAT_EQ( val1 , val2 )
```

Verifies that the two `float` values *`val1`* and *`val2`* are approximately equal, to within 4 ULPs from each other.

EXPECT_DOUBLE_EQ

```
EXPECT_DOUBLE_EQ( val1 , val2 )
ASSERT_DOUBLE_EQ( val1 , val2 )
```

Verifies that the two `double` values *`val1`* and *`val2`* are approximately equal, to within 4 ULPs from each other.

EXPECT_NEAR

```
EXPECT_NEAR( val1 , val2 , abs_error )
ASSERT_NEAR( val1 , val2 , abs_error )
```

Verifies that the difference between *`val1`* and *`val2`* does not exceed the absolute error bound *`abs_error`* .

Exception Assertions

The following assertions verify that a piece of code throws, or does not throw, an exception. Usage requires exceptions to be enabled in the build environment.

Note that the piece of code under test can be a compound statement, for example:

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

EXPECT_THROW

```
EXPECT_THROW( statement , exception_type )
ASSERT_THROW( statement , exception_type )
```

Verifies that *statement* throws an exception of type *exception_type*.

EXPECT_ANY_THROW

```
EXPECT_ANY_THROW( statement )
ASSERT_ANY_THROW( statement )
```

Verifies that *statement* throws an exception of any type.

EXPECT_NO_THROW

```
EXPECT_NO_THROW( statement )
ASSERT_NO_THROW( statement )
```

Verifies that *statement* does not throw any exception.

Predicate Assertions

The following assertions enable more complex predicates to be verified while printing a more clear failure message than if `EXPECT_TRUE` were used alone.

EXPECT_PRED*

```
EXPECT_PRED1( pred , val1 )
EXPECT_PRED2( pred , val1 , val2 )
EXPECT_PRED3( pred , val1 , val2 , val3 )
EXPECT_PRED4( pred , val1 , val2 , val3 , val4 )
EXPECT_PRED5( pred , val1 , val2 , val3 , val4 , val5 )

ASSERT_PRED1( pred , val1 )
ASSERT_PRED2( pred , val1 , val2 )
ASSERT_PRED3( pred , val1 , val2 , val3 )
ASSERT_PRED4( pred , val1 , val2 , val3 , val4 )
ASSERT_PRED5( pred , val1 , val2 , val3 , val4 , val5 )
```

Verifies that the predicate *pred* returns `true` when passed the given values as arguments.

The parameter *pred* is a function or functor that accepts as many arguments as the corresponding macro accepts values. If *pred* returns `true` for the given arguments, the assertion succeeds, otherwise the assertion fails.

When the assertion fails, it prints the value of each argument. Arguments are always evaluated exactly once.

As an example, see the following code:

```
// Returns true if m and n have no common divisors except 1.
bool MutuallyPrime(int m, int n) { ... }
...
const int a = 3;
const int b = 4;
const int c = 10;
...
```

```
EXPECT_PRED2(MutuallyPrime, a, b);  // Succeeds
EXPECT_PRED2(MutuallyPrime, b, c);  // Fails
```

In the above example, the first assertion succeeds, and the second fails with the following message:

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

Note that if the given predicate is an overloaded function or a function template, the assertion macro might not be able to determine which version to use, and it might be necessary to explicitly specify the type of the function. For example, for a Boolean function `IsPositive()` overloaded to take either a single `int` or `double` argument, it would be necessary to write one of the following:

```
EXPECT_PRED1(static_cast<bool (*)(int)>(IsPositive), 5);
EXPECT_PRED1(static_cast<bool (*)(double)>(IsPositive), 3.14);
```

Writing simply `EXPECT_PRED1(IsPositive, 5);` would result in a compiler error. Similarly, to use a template function, specify the template arguments:

```
template <typename T>
bool IsNegative(T x) {
  return x < 0;
}
...
EXPECT_PRED1(IsNegative<int>, -5);  // Must specify type for IsNegative
```

If a template has multiple parameters, wrap the predicate in parentheses so the macro arguments are parsed correctly:

```
ASSERT_PRED2((MyPredicate<int, int>), 5, 0);
```

EXPECT_PRED_FORMAT*

```
EXPECT_PRED_FORMAT1( pred_formatter , val1 )
EXPECT_PRED_FORMAT2( pred_formatter , val1 , val2 )
EXPECT_PRED_FORMAT3( pred_formatter , val1 , val2 , val3 )
EXPECT_PRED_FORMAT4( pred_formatter , val1 , val2 , val3 , val4 )
EXPECT_PRED_FORMAT5( pred_formatter , val1 , val2 , val3 , val4 , val5 )

ASSERT_PRED_FORMAT1( pred_formatter , val1 )
ASSERT_PRED_FORMAT2( pred_formatter , val1 , val2 )
ASSERT_PRED_FORMAT3( pred_formatter , val1 , val2 , val3 )
ASSERT_PRED_FORMAT4( pred_formatter , val1 , val2 , val3 , val4 )
ASSERT_PRED_FORMAT5( pred_formatter , val1 , val2 , val3 , val4 , val5 )
```

Verifies that the predicate *`pred_formatter`* succeeds when passed the given values as arguments.

The parameter *`pred_formatter`* is a *predicate-formatter*, which is a function or functor with the signature:

```
testing::AssertionResult PredicateFormatter(const char* expr1,
                                            const char* expr2,
                                            ...
                                            const char* exprn,
                                            T1 val1,
                                            T2 val2,
                                            ...
                                            Tn valn);
```

where *`val1`*, *`val2`*, ..., *`valn`* are the values of the predicate arguments, and *`expr1`*, *`expr2`*, ..., *`exprn`* are the corresponding expressions as they appear in the source code. The types `T1`, `T2`, ..., `Tn` can be either value types or reference types; if an argument has type `T`, it can be declared as either `T` or `const T&`, whichever is appropriate. For more about the return type `testing::AssertionResult`, see Using a Function That Returns an AssertionResult.

As an example, see the following code:

```
// Returns the smallest prime common divisor of m and n,
// or 1 when m and n are mutually prime.
int SmallestPrimeCommonDivisor(int m, int n) { ... }

// Returns true if m and n have no common divisors except 1.
bool MutuallyPrime(int m, int n) { ... }

// A predicate-formatter for asserting that two integers are mutually prime.
testing::AssertionResult AssertMutuallyPrime(const char* m_expr,
                                             const char* n_expr,
                                             int m,
                                             int n) {
  if (MutuallyPrime(m, n)) return testing::AssertionSuccess();

  return testing::AssertionFailure() << m_expr << " and " << n_expr
      << " (" << m << " and " << n << ") are not mutually prime, "
      << "as they have a common divisor " << SmallestPrimeCommonDivisor(m, n);
}

...
const int a = 3;
const int b = 4;
const int c = 10;
...
EXPECT_PRED_FORMAT2(AssertMutuallyPrime, a, b);  // Succeeds
EXPECT_PRED_FORMAT2(AssertMutuallyPrime, b, c);  // Fails
```

In the above example, the final assertion fails and the predicate-formatter produces the following failure message:

```
b and c (4 and 10) are not mutually prime, as they have a common divisor 2
```

Windows HRESULT Assertions

The following assertions test for `HRESULT` success or failure. For example:

```
CComPtr<IShellDispatch2> shell;
ASSERT_HRESULT_SUCCEEDED(shell.CoCreateInstance(L"Shell.Application"));
CComVariant empty;
ASSERT_HRESULT_SUCCEEDED(shell->ShellExecute(CComBSTR(url), empty, empty, empty, empty));
```

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

EXPECT_HRESULT_SUCCEEDED

`EXPECT_HRESULT_SUCCEEDED(` *`expression`* `)`
`ASSERT_HRESULT_SUCCEEDED(` *`expression`* `)`

Verifies that *`expression`* is a success `HRESULT`.

EXPECT_HRESULT_FAILED

`EXPECT_HRESULT_FAILED(` *`expression`* `)`
`ASSERT_HRESULT_FAILED(` *`expression`* `)`

Verifies that *`expression`* is a failure `HRESULT`.

Death Assertions

The following assertions verify that a piece of code causes the process to terminate. For context, see Death Tests.

These assertions spawn a new process and execute the code under test in that process. How that happens depends on the platform and the variable `::testing::GTEST_FLAG(death_test_style)`, which is initialized from the command-line flag `--gtest_death_test_style`.

- On POSIX systems, `fork()` (or `clone()` on Linux) is used to spawn the child, after which:
  - If the variable's value is `"fast"`, the death test statement is immediately executed.

    - If the variable's value is `"threadsafe"`, the child process re-executes the unit test binary just as it was originally invoked, but with some extra flags to cause just the single death test under consideration to be run.
- On Windows, the child is spawned using the `CreateProcess()` API, and re-executes the binary to cause just the single death test under consideration to be run - much like the `"threadsafe"` mode on POSIX.

Other values for the variable are illegal and will cause the death test to fail. Currently, the flag's default value is **`"fast"`**.

If the death test statement runs to completion without dying, the child process will nonetheless terminate, and the assertion fails.

Note that the piece of code under test can be a compound statement, for example:

```
EXPECT_DEATH({
  int n = 5;
  DoSomething(&n);
}, "Error on line .* of DoSomething()");
```

EXPECT_DEATH

`EXPECT_DEATH(` *`statement`* `,` *`matcher`* `)`
`ASSERT_DEATH(` *`statement`* `,` *`matcher`* `)`

Verifies that *`statement`* causes the process to terminate with a nonzero exit status and produces `stderr` output that matches *`matcher`*.

The parameter *`matcher`* is either a matcher for a `const std::string&`, or a regular expression (see Regular Expression Syntax)—a bare string *`s`* (with no matcher) is treated as `ContainsRegex(s)`, **not** `Eq(s)`.

For example, the following code verifies that calling `DoSomething(42)` causes the process to die with an error message that contains the text `My error`:

```
EXPECT_DEATH(DoSomething(42), "My error");
```

EXPECT_DEATH_IF_SUPPORTED

`EXPECT_DEATH_IF_SUPPORTED(` *`statement`* `,` *`matcher`* `)`
`ASSERT_DEATH_IF_SUPPORTED(` *`statement`* `,` *`matcher`* `)`

If death tests are supported, behaves the same as `EXPECT_DEATH`. Otherwise, verifies nothing.

EXPECT_DEBUG_DEATH

`EXPECT_DEBUG_DEATH(` *`statement`* `,` *`matcher`* `)`
`ASSERT_DEBUG_DEATH(` *`statement`* `,` *`matcher`* `)`

In debug mode, behaves the same as `EXPECT_DEATH`. When not in debug mode (i.e. `NDEBUG` is defined), just executes *`statement`*.

EXPECT_EXIT

`EXPECT_EXIT(` *`statement`* `,` *`predicate`* `,` *`matcher`* `)`
`ASSERT_EXIT(` *`statement`* `,` *`predicate`* `,` *`matcher`* `)`

Verifies that *`statement`* causes the process to terminate with an exit status that satisfies *`predicate`*, and produces `stderr` output that matches *`matcher`*.

The parameter *`predicate`* is a function or functor that accepts an `int` exit status and returns a `bool`. GoogleTest provides two predicates to handle common cases:

```
// Returns true if the program exited normally with the given exit status code.
::testing::ExitedWithCode(exit_code);

// Returns true if the program was killed by the given signal.
// Not available on Windows.
::testing::KilledBySignal(signal_number);
```

The parameter *matcher* is either a matcher for a `const std::string&`, or a regular expression (see Regular Expression Syntax)—a bare string *s* (with no matcher) is treated as `ContainsRegex(s)`, **not** `Eq(s)`.

For example, the following code verifies that calling `NormalExit()` causes the process to print a message containing the text `Success` to `stderr` and exit with exit status code 0:

```
EXPECT_EXIT(NormalExit(), testing::ExitedWithCode(0), "Success");
```