



googlemock

Google C++ Mocking Framework

 Search projects[Project Home](#) [Downloads](#) [Wiki](#) [Issues](#) [Source](#)Search Current pages ▼ for Search

CheatSheet

Google C++ Mocking Framework Cheat Sheet

Updated Aug 6, 2013 by [w...@google.com](#)

- [Defining a Mock Class](#)
 - [Mocking a Normal Class](#)
 - [Mocking a Class Template](#)
 - [Specifying Calling Conventions for Mock Functions](#)
- [Using Mocks in Tests](#)
- [Setting Default Actions](#)
- [Setting Expectations](#)
- [Matchers](#)
 - [Wildcard](#)
 - [Generic Comparison](#)
 - [Floating-Point Matchers](#)
 - [String Matchers](#)
 - [Container Matchers](#)
 - [Member Matchers](#)
 - [Matching the Result of a Function or Functor](#)
 - [Pointer Matchers](#)
 - [Multiargument Matchers](#)
 - [Composite Matchers](#)
 - [Adapters for Matchers](#)
 - [Matchers as Predicates](#)
 - [Defining Matchers](#)
 - [Matchers as Test Assertions](#)
- [Actions](#)
 - [Returning a Value](#)
 - [Side Effects](#)
 - [Using a Function or a Functor as an Action](#)
 - [Default Action](#)
 - [Composite Actions](#)
 - [Defining Actions](#)
- [Cardinalities](#)
- [Expectation Order](#)
 - [The After Clause](#)
 - [Sequences](#)
- [Verifying and Resetting a Mock](#)
- [Mock Classes](#)
- [Flags](#)

Defining a Mock Class

Mocking a Normal Class

Given

```
class Foo {
...
virtual ~Foo();
virtual int GetSize() const = 0;
virtual string Describe(const char* name) = 0;
virtual string Describe(int type) = 0;
virtual bool Process(Bar elem, int count) = 0;
```

```
};
```

(note that ~Foo() **must** be virtual) we can define its mock as

```
#include "gmock/gmock.h"

class MockFoo : public Foo {
    MOCK_CONST_METHOD0(GetSize, int());
    MOCK_METHOD1(Describe, string(const char* name));
    MOCK_METHOD1(Describe, string(int type));
    MOCK_METHOD2(Process, bool(Bar elem, int count));
};
```

To create a "nice" mock object which ignores all uninteresting calls, or a "strict" mock object, which treats them as failures:

```
NiceMock<MockFoo> nice_foo;    // The type is a subclass of MockFoo.
StrictMock<MockFoo> strict_foo; // The type is a subclass of MockFoo.
```

Mocking a Class Template

To mock

```
template <typename Elem>
class StackInterface {
public:
    ...
    virtual ~StackInterface();
    virtual int GetSize() const = 0;
    virtual void Push(const Elem& x) = 0;
};
```

(note that ~StackInterface() **must** be virtual) just append _T to the MOCK_* macros:

```
template <typename Elem>
class MockStack : public StackInterface<Elem> {
public:
    ...
    MOCK_CONST_METHOD0_T(GetSize, int());
    MOCK_METHOD1_T(Push, void(const Elem& x));
};
```

Specifying Calling Conventions for Mock Functions

If your mock function doesn't use the default calling convention, you can specify it by appending _WITH_CALLTYPE to any of the macros described in the previous two sections and supplying the calling convention as the first argument to the macro. For example,

```
MOCK_METHOD1_WITH_CALLTYPE(STDMETHODCALLTYPE, Foo, bool(int n));
MOCK_CONST_METHOD2_WITH_CALLTYPE(STDMETHODCALLTYPE, Bar, int(double x, double y));
```

where STDMETHODCALLTYPE is defined by <objbase.h> on Windows.

Using Mocks in Tests

The typical flow is:

1. Import the Google Mock names you need to use. All Google Mock names are in the testing namespace unless they are macros or otherwise noted.
2. Create the mock objects.
3. Optionally, set the default actions of the mock objects.
4. Set your expectations on the mock objects (How will they be called? What will they do?).
5. Exercise code that uses the mock objects; if necessary, check the result using [Google Test](#) assertions.
6. When a mock object is destructed, Google Mock automatically verifies that all expectations on it have been satisfied.

Here is an example:

```
using ::testing::Return; // #1

TEST(BarTest, DoesThis) {
    MockFoo foo; // #2
```

```

EXPECT_CALL(foo, ...) // #2
ON_CALL(foo, GetSize()) // #3
    .WillByDefault(Return(1));
// ... other default actions ...

EXPECT_CALL(foo, Describe(5)) // #4
    .Times(3)
    .WillRepeatedly(Return("Category 5"));
// ... other expectations ...

EXPECT_EQ("good", MyProductionFunction(&foo)); // #5
} // #6

```

Setting Default Actions

Google Mock has a **built-in default action** for any function that returns void, bool, a numeric value, or a pointer.

To customize the default action for functions with return type T globally:

```

using ::testing::DefaultValue;

DefaultValue<T>::Set(value); // Sets the default value to be returned.
// ... use the mocks ...
DefaultValue<T>::Clear(); // Resets the default value.

```

To customize the default action for a particular method, use ON_CALL():

```

ON_CALL(mock_object, method(matchers))
    .With(multi_argument_matcher) ?
    .WillByDefault(action);

```

Setting Expectations

EXPECT_CALL() sets **expectations** on a mock method (How will it be called? What will it do?):

```

EXPECT_CALL(mock_object, method(matchers))
    .With(multi_argument_matcher) ?
    .Times(cardinality) ?
    .InSequence(sequences) *
    .After(expectations) *
    .WillOnce(action) *
    .WillRepeatedly(action) ?
    .RetiresOnSaturation(); ?

```

If Times() is omitted, the cardinality is assumed to be:

- Times(1) when there is neither WillOnce() nor WillRepeatedly();
- Times(n) when there are n WillOnce()s but no WillRepeatedly(), where n >= 1; or
- Times(AtLeast(n)) when there are n WillOnce()s and a WillRepeatedly(), where n >= 0.

A method with no EXPECT_CALL() is free to be invoked *any number of times*, and the default action will be taken each time.

Matchers

A **matcher** matches a *single* argument. You can use it inside ON_CALL() or EXPECT_CALL(), or use it to validate a value directly:

EXPECT_THAT(value, matcher)	Asserts that value matches matcher.
ASSERT_THAT(value, matcher)	The same as EXPECT_THAT(value, matcher), except that it generates a fatal failure.

Built-in matchers (where argument is the function argument) are divided into several categories:

Wildcard

_	argument can be any value of the correct type.
A<type>() or An<type>()	argument can be any value of type type.

Generic Comparisons

Generic Comparison

Eq(value) or value	argument == value
Ge(value)	argument >= value
Gt(value)	argument > value
Le(value)	argument <= value
Lt(value)	argument < value
Ne(value)	argument != value
IsNull()	argument is a NULL pointer (raw or smart).
NotNull()	argument is a non-null pointer (raw or smart).
Ref(variable)	argument is a reference to variable.
TypedEq<type>(value)	argument has type type and is equal to value. You may need to use this instead of Eq(value) when the mock function is overloaded.

Except Ref(), these matchers make a *copy* of value in case it's modified or destructed later. If the compiler complains that value doesn't have a public copy constructor, try wrap it in ByRef(), e.g. Eq(ByRef(non_copyable_value)). If you do that, make sure non_copyable_value is not changed afterwards, or the meaning of your matcher will be changed.

Floating-Point Matchers

DoubleEq(a_double)	argument is a double value approximately equal to a_double, treating two NaNs as unequal.
FloatEq(a_float)	argument is a float value approximately equal to a_float, treating two NaNs as unequal.
NanSensitiveDoubleEq(a_double)	argument is a double value approximately equal to a_double, treating two NaNs as equal.
NanSensitiveFloatEq(a_float)	argument is a float value approximately equal to a_float, treating two NaNs as equal.

The above matchers use ULP-based comparison (the same as used in [Google Test](#)). They automatically pick a reasonable error bound based on the absolute value of the expected value. DoubleEq() and FloatEq() conform to the IEEE standard, which requires comparing two NaNs for equality to return false. The NanSensitive* version instead treats two NaNs as equal, which is often what a user wants.

DoubleNear(a_double, max_abs_error)	argument is a double value close to a_double (absolute error <= max_abs_error), treating two NaNs as unequal.
FloatNear(a_float, max_abs_error)	argument is a float value close to a_float (absolute error <= max_abs_error), treating two NaNs as unequal.
NanSensitiveDoubleNear(a_double, max_abs_error)	argument is a double value close to a_double (absolute error <= max_abs_error), treating two NaNs as equal.
NanSensitiveFloatNear(a_float, max_abs_error)	argument is a float value close to a_float (absolute error <= max_abs_error), treating two NaNs as equal.

String Matchers

The argument can be either a C string or a C++ string object:

ContainsRegex(string)	argument matches the given regular expression.
EndsWith(suffix)	argument ends with string suffix.
HasSubstr(string)	argument contains string as a sub-string.
MatchesRegex(string)	argument matches the given regular expression with the match starting at the first character and ending at the last character.
StartsWith(prefix)	argument starts with string prefix.
StrCaseEq(string)	argument is equal to string, ignoring case.
StrCaseNe(string)	argument is not equal to string, ignoring case.
StrEq(string)	argument is equal to string.
StrNe(string)	argument is not equal to string.

ContainsRegex() and MatchesRegex() use the regular expression syntax defined [here](#). StrCaseEq(), StrCaseNe(), StrEq(), and StrNe() work for wide strings as well.

Container Matchers

Most STL-style containers support ==, so you can use Eq(expected_container) or simply expected_container to match a container exactly. If you want to write the elements in-line, match them more flexibly, or get more informative messages, you can use:

ContainerEq(container)	The same as Eq(container) except that the failure message also includes which elements are in one container but not the other.
Contains(e)	argument contains an element that matches e, which can be either a value or a matcher.
Each(e)	argument is a container where <i>every</i> element matches e, which can be either a value or a matcher.
ElementsAre(e0, e1, ..., en)	argument has n + 1 elements, where the i-th element matches ei, which can be a value or a matcher. 0 to 10 arguments are allowed.
ElementsAreArray({ e0, e1, ..., en }), ElementsAreArray(array), or ElementsAreArray(array, count)	The same as ElementsAre() except that the expected element values/matchers come from an initializer list, vector, or C-style array.
IsEmpty()	argument is an empty container (container.empty()).
Pointwise(m, container)	argument contains the same number of elements as in container, and for all i, (the i-th element in argument, the i-th element in container) match m, which is a matcher on 2-tuples. E.g. Pointwise(Le(), upper_bounds) verifies that each element in argument doesn't exceed the corresponding element in upper_bounds.
SizeIs(m)	argument is a container whose size matches m. E.g. SizeIs(2) or SizeIs(Lt(2)).
UnorderedElementsAre(e0, e1, ..., en)	argument has n + 1 elements, and under some permutation each element matches an ei (for a different i), which can be a value or a matcher. 0 to 10 arguments are allowed.
UnorderedElementsAreArray({ e0, e1, ..., en }), UnorderedElementsAreArray(array), or UnorderedElementsAreArray(array, count)	The same as UnorderedElementsAre() except that the expected element values/matchers come from an initializer list, vector, or C-style array.
WhenSorted(m)	When argument is sorted using the < operator, it matches container matcher m. E.g. WhenSorted(UnorderedElementsAre(1, 2, 3)) verifies that argument contains elements 1, 2, and 3, ignoring order.
WhenSortedBy(comparator, m)	The same as WhenSorted(m), except that the given comparator instead of < is used to sort argument. E.g. WhenSortedBy(std::greater<int>(), ElementsAre(3, 2, 1)).

These matchers can also match:

1. a native array passed by reference (e.g. in Foo(const int (&a)[5])), and
2. an array passed as a pointer and a count (e.g. in Bar(const T* buffer, int len) -- see [Multi-argument Matchers](#)).

where the array may be multi-dimensional (i.e. its elements can be arrays).

Member Matchers

Field(&class::field, m)	argument.field (or argument->field when argument is a plain pointer) matches matcher m, where argument is an object of type class.
Key(e)	argument.first matches e, which can be either a value or a matcher. E.g. Contains(Key(Le(5))) can verify that a map contains a key <= 5.
Pair(m1, m2)	argument is an std::pair whose first field matches m1 and second field matches m2.
Property(&class::property, m)	argument.property() (or argument->property() when argument is a plain pointer) matches matcher m, where argument is an object of type class.

Matching the Result of a Function or Functor

ResultOf(f, m)	f(argument) matches matcher m, where f is a function or functor.
----------------	--

Pointer Matchers

Pointee(m)	argument (either a smart pointer or a raw pointer) points to a value that matches matcher m.
------------	--

Multiargument Matchers

Technically, all matchers match a *single* value. A "multi-argument" matcher is just one that matches a *tuple*. The following matchers can be used to match a tuple (x, y):

Eq()	x == y
Ge()	x >= y
Gt()	x > y
Le()	x <= y
Lt()	x < y
Ne()	x != y

You can use the following selectors to pick a subset of the arguments (or reorder them) to participate in the matching:

AllArgs(m)	Equivalent to m. Useful as syntactic sugar in .With(AllArgs(m)).
Args<N1, N2, ..., Nk>(m)	The tuple of the k selected (using 0-based indices) arguments matches m, e.g. Args<1, 2>(Eq()).

Composite Matchers

You can make a matcher from one or more other matchers:

AllOf(m1, m2, ..., mn)	argument matches all of the matchers m1 to mn.
AnyOf(m1, m2, ..., mn)	argument matches at least one of the matchers m1 to mn.
Not(m)	argument doesn't match matcher m.

Adapters for Matchers

MatcherCast<T>(m)	casts matcher m to type Matcher<T>.
SafeMatcherCast<T>(m)	safely casts matcher m to type Matcher<T>.
Truly(predicate)	predicate(argument) returns something considered by C++ to be true, where predicate is a function or functor.

Matchers as Predicates

Matches(m)(value)	evaluates to true if value matches m. You can use Matches(m) alone as a unary functor.
ExplainMatchResult(m, value, result_listener)	evaluates to true if value matches m, explaining the result to result_listener.
Value(value, m)	evaluates to true if value matches m.

Defining Matchers

MATCHER(IsEven, "") { return (arg % 2) == 0; }	Defines a matcher IsEven() to match an even number.
MATCHER_P(IsDivisibleBy, n, "") { *result_listener << "where the remainder is " << (arg % n); return (arg % n) == 0; }	Defines a macher IsDivisibleBy(n) to match a number divisible by n.
MATCHER_P2(IsBetween, a, b, std::string(negation ? "isn't" : "is") + " between " + PrintToString(a) + " and " + PrintToString(b)) { return a <= arg && arg <= b; }	Defines a matcher IsBetween(a, b) to match a value in the range [a, b].

Notes:

1. The MATCHER* macros cannot be used inside a function or class.
2. The matcher body must be *purely functional* (i.e. it cannot have any side effect, and the result must not depend on anything other than the value being matched and the matcher parameters).
3. You can use PrintToString(x) to convert a value x of any type to a string.

Matchers as Test Assertions

ASSERT_THAT(expression, m)	Generates a fatal failure if the value of expression doesn't match matcher m.
EXPECT_THAT(expression, m)	Generates a non-fatal failure if the value of expression doesn't match matcher m.

Actions

Actions specify what a mock function should do when invoked.

Returning a Value

Return()	Return from a void mock function.
Return(value)	Return value. If the type of value is different to the mock function's return type, value is converted to the latter type <i>at the time the expectation is set</i> , not when the action is executed.
ReturnArg<N>()	Return the N-th (0-based) argument.
ReturnNew<T>(a1, ..., ak)	Return new T(a1, ..., ak); a different object is created each time.
ReturnNull()	Return a null pointer.
ReturnPointee(ptr)	Return the value pointed to by ptr.
ReturnRef(variable)	Return a reference to variable.
ReturnRefOfCopy(value)	Return a reference to a copy of value; the copy lives as long as the action.

Side Effects

Assign(&variable, value)	Assign value to variable.
DeleteArg<N>()	Delete the N-th (0-based) argument, which must be a pointer.
SaveArg<N>(pointer)	Save the N-th (0-based) argument to *pointer.
SaveArgPointee<N>(pointer)	Save the value pointed to by the N-th (0-based) argument to *pointer.
SetArgReferee<N>(value)	Assign value to the variable referenced by the N-th (0-based) argument.
SetArgPointee<N>(value)	Assign value to the variable pointed by the N-th (0-based) argument.
SetArgumentPointee<N>(value)	Same as SetArgPointee<N>(value). Deprecated. Will be removed in v1.7.0.
SetArrayArgument<N>(first, last)	Copies the elements in source range [first, last) to the array pointed to by the N-th (0-based) argument, which can be either a pointer or an iterator. The action does not take ownership of the elements in the source range.
SetErrnoAndReturn(error, value)	Set errno to error and return value.
Throw(exception)	Throws the given exception, which can be any copyable value. Available since v1.1.0.

Using a Function or a Functor as an Action

Invoke(f)	Invoke f with the arguments passed to the mock function, where f can be a global/static function or a functor.
Invoke(object_pointer, &class::method)	Invoke the {method on the object with the arguments passed to the mock function.
InvokeWithoutArgs(f)	Invoke f, which can be a global/static function or a functor. f must take no arguments.
InvokeWithoutArgs(object_pointer, &class::method)	Invoke the method on the object, which takes no arguments.
InvokeArgument<N>(arg1, arg2, ..., argk)	Invoke the mock function's N-th (0-based) argument, which must be a function or a functor, with the k arguments.

The return value of the invoked function is used as the return value of the action.

When defining a function or functor to be used with `Invoke*()`, you can declare any unused parameters as `Unused`:

```
double Distance(Unused, double x, double y) { return sqrt(x*x + y*y); }
...
EXPECT_CALL(mock, Foo("Hi", _, _)).WillOnce(Invoke(Distance));
```

In `InvokeArgument<N>(...)`, if an argument needs to be passed by reference, wrap it inside `ByRef()`. For example,

```
InvokeArgument<2>(5, string("Hi"), ByRef(foo))
```

calls the mock function's #2 argument, passing to it 5 and `string("Hi")` by value, and `foo` by reference.

Default Action

<code>DoDefault()</code>	Do the default action (specified by <code>ON_CALL()</code> or the built-in one).
--------------------------	--

Note: due to technical reasons, `DoDefault()` cannot be used inside a composite action - trying to do so will result in a run-time error.

Composite Actions

<code>DoAll(a1, a2, ..., an)</code>	Do all actions <code>a1</code> to <code>an</code> and return the result of <code>an</code> in each invocation. The first <code>n - 1</code> sub-actions must return void.
<code>IgnoreResult(a)</code>	Perform action <code>a</code> and ignore its result. <code>a</code> must not return void.
<code>WithArg<N>(a)</code>	Pass the <code>N</code> -th (0-based) argument of the mock function to action <code>a</code> and perform it.
<code>WithArgs<N1, N2, ..., Nk>(a)</code>	Pass the selected (0-based) arguments of the mock function to action <code>a</code> and perform it.
<code>WithoutArgs(a)</code>	Perform action <code>a</code> without any arguments.

Defining Actions

<code>ACTION(Sum) { return arg0 + arg1; }</code>	Defines an action <code>Sum()</code> to return the sum of the mock function's argument #0 and #1.
<code>ACTION_P(Plus, n) { return arg0 + n; }</code>	Defines an action <code>Plus(n)</code> to return the sum of the mock function's argument #0 and <code>n</code> .
<code>ACTION_Pk(Foo, p1, ..., pk) { statements; }</code>	Defines a parameterized action <code>Foo(p1, ..., pk)</code> to execute the given statements.

The `ACTION*` macros cannot be used inside a function or class.

Cardinalities

These are used in `Times()` to specify how many times a mock function will be called:

<code>AnyNumber()</code>	The function can be called any number of times.
<code>AtLeast(n)</code>	The call is expected at least <code>n</code> times.
<code>AtMost(n)</code>	The call is expected at most <code>n</code> times.
<code>Between(m, n)</code>	The call is expected between <code>m</code> and <code>n</code> (inclusive) times.
<code>Exactly(n)</code> or <code>n</code>	The call is expected exactly <code>n</code> times. In particular, the call should never happen when <code>n</code> is 0.

Expectation Order

By default, the expectations can be matched in *any* order. If some or all expectations must be matched in a given order, there are two ways to specify it. They can be used either independently or together.

The After Clause

```
using ::testing::Expectation;
...
Expectation init_x = EXPECT_CALL(foo, InitX());
Expectation init_y = EXPECT_CALL(foo, InitY());
EXPECT_CALL(foo, Bar());
```



```
EXPECT_CALL(foo, Bar())
  .After(init_x, init_y);
```

says that `Bar()` can be called only after both `InitX()` and `InitY()` have been called.

If you don't know how many pre-requisites an expectation has when you write it, you can use an `ExpectationSet` to collect them:

```
using ::testing::ExpectationSet;
...
ExpectationSet all_inits;
for (int i = 0; i < element_count; i++) {
  all_inits += EXPECT_CALL(foo, InitElement(i));
}
EXPECT_CALL(foo, Bar())
  .After(all_inits);
```

says that `Bar()` can be called only after all elements have been initialized (but we don't care about which elements get initialized before the others).

Modifying an `ExpectationSet` after using it in an `.After()` doesn't affect the meaning of the `.After()`.

Sequences

When you have a long chain of sequential expectations, it's easier to specify the order using **sequences**, which don't require you to give each expectation in the chain a different name. *All expected calls* in the same sequence must occur in the order they are specified.

```
using ::testing::Sequence;
Sequence s1, s2;
...
EXPECT_CALL(foo, Reset())
  .InSequence(s1, s2)
  .WillOnce(Return(true));
EXPECT_CALL(foo, GetSize())
  .InSequence(s1)
  .WillOnce(Return(1));
EXPECT_CALL(foo, Describe(A<const char*>()))
  .InSequence(s2)
  .WillOnce(Return("dummy"));
```

says that `Reset()` must be called before *both* `GetSize()` and `Describe()`, and the latter two can occur in any order.

To put many expectations in a sequence conveniently:

```
using ::testing::InSequence;
{
  InSequence dummy;

  EXPECT_CALL(...)...;
  EXPECT_CALL(...)...;
  ...
  EXPECT_CALL(...)...;
}
```

says that all expected calls in the scope of `dummy` must occur in strict order. The name `dummy` is irrelevant.)

Verifying and Resetting a Mock

Google Mock will verify the expectations on a mock object when it is destructed, or you can do it earlier:

```
using ::testing::Mock;
...
// Verifies and removes the expectations on mock_obj;
// returns true iff successful.
Mock::VerifyAndClearExpectations(&mock_obj);
...
// Verifies and removes the expectations on mock_obj;
// also removes the default actions set by ON_CALL();
// returns true iff successful.
Mock::VerifyAndClear(mock_obj);
```

You can also tell Google Mock that a mock object can be leaked and doesn't need to be verified:

```
Mock::AllowLeak(&mock_obj);
```

Mock Classes

Google Mock defines a convenient mock class template

```
class MockFunction<R(A1, ..., An)> {  
public:  
    MOCK_METHODn(Call, R(A1, ..., An));  
};
```

See this [recipe](#) for one application of it.

Flags

<code>--gmock_catch_leaked_mocks=0</code>	Don't report leaked mock objects as failures.
<code>--gmock_verbose=LEVEL</code>	Sets the default verbosity level (info, warning, or error) of Google Mock messages.

[Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)