invalid-email-address Googletest export

11d9834 on Feb 7

5 contributors

Raw Blame History

776 lines (613 sloc) 36 KB

gMock Cheat Sheet

Defining a Mock Class

Mocking a Normal Class {#MockClass}

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_METHOD(int, GetSize, (), (const, override));
  MOCK_METHOD(string, Describe, (const char* name), (override));
  MOCK_METHOD(string, Describe, (int type), (override));
  MOCK_METHOD(bool, Process, (Bar elem, int count), (override));
};
```

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

```
using ::testing::NiceMock;
using ::testing::NaggyMock;
using ::testing::StrictMock;

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

Note: A mock object is currently naggy by default. We may make it nice by default in the future.

Mocking a Class Template {#MockTemplate}

Class templates can be mocked just like any class.

To mock

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

(note that all member functions that are mocked, including `~StackInterface()` **must** be virtual).

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

Specifying Calling Conventions for Mock Functions

If your mock function doesn't use the default calling convention, you can specify it by adding `Calltype(convention)` to `MOCK_METHOD`'s 4th parameter. For example,

```
  MOCK_METHOD(bool, Foo, (int n), (Calltype(STDMETHODCALLTYPE)));
  MOCK_METHOD(int, Bar, (double x, double y),
              (const, Calltype(STDMETHODCALLTYPE)));
```

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

Using Mocks in Tests {#UsingMocks}

The typical work flow is:

1. Import the gMock names you need to use. All gMock symbols 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 googletest assertions.

6. When a mock object is destructed, gMock automatically verifies that all expectations on it have been satisfied.

Here's an example:

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

TEST(BarTest, DoesThis) {
  MockFoo 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 {#OnCall}

gMock has a **built-in default action** for any function that returns `void`, `bool`, a numeric value, or a pointer. In C++11, it will additionally returns the default-constructed value, if one exists for the given type.

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

```
using ::testing::DefaultValue;

// Sets the default value to be returned. T must be CopyConstructible.
DefaultValue<T>::Set(value);
// Sets a factory. Will be invoked on demand. T must be MoveConstructible.
//  T MakeT();
DefaultValue<T>::SetFactory(&MakeT);
// ... use the mocks ...
// Resets the default value.
DefaultValue<T>::Clear();
```

Example usage:

```
  // Sets the default action for return type std::unique_ptr<Buzz> to
  // creating a new Buzz every time.
  DefaultValue<std::unique_ptr<Buzz>>::SetFactory(
      [] { return MakeUnique<Buzz>(AccessLevel::kInternal); });

  // When this fires, the default action of MakeBuzz() will run, which
  // will return a new Buzz object.
  EXPECT_CALL(mock_buzzer_, MakeBuzz("hello")).Times(AnyNumber());

  auto buzz1 = mock_buzzer_.MakeBuzz("hello");
  auto buzz2 = mock_buzzer_.MakeBuzz("hello");
  EXPECT_NE(nullptr, buzz1);
  EXPECT_NE(nullptr, buzz2);
  EXPECT_NE(buzz1, buzz2);

  // Resets the default action for return type std::unique_ptr<Buzz>,
  // to avoid interfere with other tests.
  DefaultValue<std::unique_ptr<Buzz>>::Clear();
```

To customize the default action for a particular method of a specific mock object, use `ON_CALL()`. `ON_CALL()` has a similar syntax to `EXPECT_CALL()`, but it is used for setting default behaviors (when you do not require that the mock method is called). See here for a more detailed discussion.

```
ON_CALL(mock-object, method(matchers))
    .With(multi-argument-matcher)   ?
    .WillByDefault(action);
```

Setting Expectations {#ExpectCall}

`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();        ?
```

For each item above, `?` means it can be used at most once, while `*` means it can be used any number of times.

In order to pass, `EXPECT_CALL` must be used before the calls are actually made.

The `(matchers)` is a comma-separated list of matchers that correspond to each of the arguments of `method`, and sets the expectation only for calls of `method` that matches all of the matchers.

If `(matchers)` is omitted, the expectation is the same as if the matchers were set to anything matchers (for example, `(_, _, _, _)` for a four-arg method).

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 {#MatcherList}

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

Macro	Description
`EXPECT_THAT(actual_value, matcher)`	Asserts that `actual_value` matches `matcher`.
`ASSERT_THAT(actual_value, matcher)`	The same as `EXPECT_THAT(actual_value, matcher)`, except that it generates a **fatal** failure.

Built-in matchers (where `argument` is the function argument, e.g. `actual_value` in the example above, or when used in the context of `EXPECT_CALL(mock_object, method(matchers))`, the arguments of `method`) are divided into several categories:

Wildcard

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

Generic Comparison

Matcher	Description
`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`
`IsFalse()`	`argument` evaluates to `false` in a Boolean context.
`IsTrue()`	`argument` evaluates to `true` in a Boolean context.
`IsNull()`	`argument` is a `NULL` pointer (raw or smart).
`NotNull()`	`argument` is a non-null pointer (raw or smart).
`Optional(m)`	`argument` is `optional<>` that contains a value matching `m`. (For testing whether an `optional<>` is set, check for equality with `nullopt`. You may need to use `Eq(nullopt)` if the inner type doesn't have `==`.)
`VariantWith<T>(m)`	`argument` is `variant<>` that holds the alternative of type T with a value matching `m`.
`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.

`IsTrue` and `IsFalse` are useful when you need to use a matcher, or for types that can be explicitly converted to Boolean, but are not implicitly converted to Boolean. In other cases, you can use the basic `EXPECT_TRUE` and `EXPECT_FALSE` assertions.

Floating-Point Matchers {#FpMatchers}

Matcher	Description
`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.

Matcher	Description
`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.
`IsNan()`	`argument` is any floating-point type with a NaN value.

The above matchers use ULP-based comparison (the same as used in googletest). 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.

Matcher	Description
`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:

Matcher	Description
`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()` take ownership of the `RE` object. They use the regular expression syntax defined here. All of these matchers, except `ContainsRegex()` and `MatchesRegex()` 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:

Matcher	Description
`BeginEndDistanceIs(m)`	`argument` is a container whose `begin()` and `end()` iterators are separated by a number of increments matching `m`. E.g. `BeginEndDistanceIs(2)` or `BeginEndDistanceIs(Lt(2))`. For containers that define a `size()` method, `SizeIs(m)` may be more efficient.
`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 *every* 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.
`ElementsAreArray({e0, e1, ..., en})`, `ElementsAreArray(a_container)`, `ElementsAreArray(begin, end)`, `ElementsAreArray(array)`, or `ElementsAreArray(array, count)`	The same as `ElementsAre()` except that the expected element values/matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`IsEmpty()`	`argument` is an empty container (`container.empty()`).
`IsSubsetOf({e0, e1, ..., en})`, `IsSubsetOf(a_container)`, `IsSubsetOf(begin, end)`, `IsSubsetOf(array)`, or `IsSubsetOf(array, count)`	`argument` matches `UnorderedElementsAre(x0, x1, ..., xk)` for some subset `{x0, x1, ..., xk}` of the expected matchers.
`IsSupersetOf({e0, e1, ..., en})`, `IsSupersetOf(a_container)`, `IsSupersetOf(begin, end)`, `IsSupersetOf(array)`, or `IsSupersetOf(array, count)`	Some subset of `argument` matches `UnorderedElementsAre(` expected matchers `)`.
`Pointwise(m, container)`, `Pointwise(m, {e0, e1, ..., en})`	`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`. See more detail below.
`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 of the elements, each element matches an `ei` (for a different `i`), which can be a value or a matcher.

Matcher	Description
`UnorderedElementsAreArray({e0, e1, ..., en})`, `UnorderedElementsAreArray(a_container)`, `UnorderedElementsAreArray(begin, end)`, `UnorderedElementsAreArray(array)`, or `UnorderedElementsAreArray(array, count)`	The same as `UnorderedElementsAre()` except that the expected element values/matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`UnorderedPointwise(m, container)`, `UnorderedPointwise(m, {e0, e1, ..., en})`	Like `Pointwise(m, container)`, but ignores the order of elements.
`WhenSorted(m)`	When `argument` is sorted using the `<` operator, it matches container matcher `m`. E.g. `WhenSorted(ElementsAre(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(), ElementsAre(3, 2, 1))`.

Notes:

- These matchers can also match:
  i. a native array passed by reference (e.g. in `Foo(const int (&a)[5])`), and
  ii. an array passed as a pointer and a count (e.g. in `Bar(const T* buffer, int len)` -- see Multi-argument Matchers).
- The array being matched may be multi-dimensional (i.e. its elements can be arrays).
- `m` in `Pointwise(m, ...)` should be a matcher for `::std::tuple<T, U>` where `T` and `U` are the element type of the actual container and the expected container, respectively. For example, to compare two `Foo` containers where `Foo` doesn't support `operator==`, one might write:

```
using ::std::get;
MATCHER(FooEq, "") {
  return std::get<0>(arg).Equals(std::get<1>(arg));
}
...
EXPECT_THAT(actual_foos, Pointwise(FooEq(), expected_foos));
```

Member Matchers

Matcher	Description
`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`.

Matcher	Description
`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, Functor, or Callback

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

Pointer Matchers

Matcher	Description
`Pointee(m)`	`argument` (either a smart pointer or a raw pointer) points to a value that matches matcher `m`.
`WhenDynamicCastTo<T>(m)`	when `argument` is passed through `dynamic_cast<T>()`, it matches matcher `m`.

Multi-argument Matchers {#MultiArgMatchers}

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)`:

Matcher	Description
`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:

Matcher	Description
`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:

Matcher	Description
`AllOf(m1, m2, ..., mn)`	`argument` matches all of the matchers `m1` to `mn`.
`AllOfArray({m0, m1, ..., mn})`, `AllOfArray(a_container)`, `AllOfArray(begin, end)`, `AllOfArray(array)`, or `AllOfArray(array, count)`	The same as `AllOf()` except that the matchers come from an initializer list, STL-style container, iterator range, or C-style array.

Matcher	Description
`AnyOf(m1, m2, ..., mn)`	`argument` matches at least one of the matchers `m1` to `mn`.
`AnyOfArray({m0, m1, ..., mn})`, `AnyOfArray(a_container)`, `AnyOfArray(begin, end)`, `AnyOfArray(array)`, or `AnyOfArray(array, count)`	The same as `AnyOf()` except that the matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`Not(m)`	`argument` doesn't match matcher `m`.

Adapters for Matchers

Matcher	Description
`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.

`AddressSatisfies(callback)` and `Truly(callback)` take ownership of `callback`, which must be a permanent callback.

Using Matchers as Predicates {#MatchersAsPredicatesCheat}

Matcher	Description
`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	Description
`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 matcher `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.

Actions {#ActionList}

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 *at the time the expectation is set*, 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.
`ReturnRoundRobin({a1, ..., ak})`	Each call will return the next `ai` in the list, starting at the beginning when the end of the list is reached.

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, Functor, or Lambda as an Action

In the following, by "callable" we mean a free function, `std::function`, functor, or lambda.

`f`	Invoke f with the arguments passed to the mock function, where f is a callable.
`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 callable to be used with `Invoke*()`, you can declare any unused parameters as `Unused`:

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

`Invoke(callback)` and `InvokeWithoutArgs(callback)` take ownership of `callback`, which must be permanent. The type of `callback` must be a base callback type instead of a derived one, e.g.

```
  BlockingClosure* done = new BlockingClosure;
  ... Invoke(done) ...;  // This won't compile!

  Closure* done2 = new BlockingClosure;
  ... Invoke(done2) ...;  // This works.
```

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

```
using ::testing::ByRef;
using ::testing::InvokeArgument;
...
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

Matcher	Description
`DoDefault()`	Do the default action (specified by `ON_CALL()` 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

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

Defining Actions

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

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

Cardinalities {#CardinalityList}

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

`AnyNumber()`	The function can be called any number of times.
`AtLeast(n)`	The call is expected at least `n` times.
`AtMost(n)`	The call is expected at most `n` times.
`Between(m, n)`	The call is expected between `m` and `n` (inclusive) times.
`Exactly(n) or n`	The call is expected exactly `n` times. In particular, the call should never happen when `n` 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 {#AfterClause}

```
using ::testing::Expectation;
...
Expectation init_x = EXPECT_CALL(foo, InitX());
Expectation init_y = EXPECT_CALL(foo, InitY());
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 {#UsingSequences}

When you have a long chain of sequential expectations, it's easier to specify the order using **sequences**, which don't require you to given 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::Return;
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 seq;

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

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

Verifying and Resetting a Mock

gMock 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 if and only if successful.
Mock::VerifyAndClearExpectations(&mock_obj);
```

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

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

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

Mock Classes

gMock defines a convenient mock class template

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

See this recipe for one application of it.

Flags

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