gMock Cheat Sheet

Defining a Mock Class

Mocking a Normal Class {#MockClass}

Given

```
class Foo {
 public:
  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 {
 public:
  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.
```

{: .callout .note}
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 {
 public:
  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> {
 public:
  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(MyProductionFunction(&foo), "good");  // #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`, use `DefaultValue<T>`. For example:

```
  // 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(buzz1, nullptr);
  EXPECT_NE(buzz2, nullptr);
  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 Knowing When to Expect for a more detailed discussion.

Setting Expectations {#ExpectCall}

See `EXPECT_CALL` in the Mocking Reference.

Matchers {#MatcherList}

See the Matchers Reference.

Actions {#ActionList}

See the Actions Reference.

Cardinalities {#CardinalityList}

See the `Times` clause of
`EXPECT_CALL` in the Mocking Reference.

Expectation Order

By default, expectations can be matched in *any* order. If some or all
expectations must be matched in a given order, you can use the
`After` clause or
`InSequence` clause of
`EXPECT_CALL`, or use an `InSequence` object.

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);
```

Do not set new expectations after verifying and clearing a mock after its use.
Setting expectations after code that exercises the mock has undefined behavior.
See Using Mocks in Tests for more
information.

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.