gMock Cookbook

You can find recipes for using gMock here. If you haven't yet, please read
the dummy guide first to make sure you understand the
basics.

{: .callout .note}
Note: gMock lives in the `testing` name space. For readability, it is
recommended to write `using ::testing::Foo;` once in your file before using the
name `Foo` defined by gMock. We omit such `using` statements in this section for
brevity, but you should do it in your own code.

Creating Mock Classes

Mock classes are defined as normal classes, using the `MOCK_METHOD` macro to
generate mocked methods. The macro gets 3 or 4 parameters:

```
class MyMock {
 public:
  MOCK_METHOD(ReturnType, MethodName, (Args...));
  MOCK_METHOD(ReturnType, MethodName, (Args...), (Specs...));
};
```

The first 3 parameters are simply the method declaration, split into 3 parts.
The 4th parameter accepts a closed list of qualifiers, which affect the
generated method:

- `const` - Makes the mocked method a `const` method. Required if
  overriding a `const` method.
- `override` - Marks the method with `override`. Recommended if overriding
  a `virtual` method.
- `noexcept` - Marks the method with `noexcept`. Required if overriding a
  `noexcept` method.
- `Calltype(...)` - Sets the call type for the method (e.g. to
  `STDMETHODCALLTYPE`), useful in Windows.
- `ref(...)` - Marks the method with the reference qualification
  specified. Required if overriding a method that has reference
  qualifications. Eg `ref(&)` or `ref(&&)`.

Dealing with unprotected commas

Unprotected commas, i.e. commas which are not surrounded by parentheses, prevent
`MOCK_METHOD` from parsing its arguments correctly:

{: .bad}

```
class MockFoo {
 public:
  MOCK_METHOD(std::pair<bool, int>, GetPair, ());  // Won't compile!
  MOCK_METHOD(bool, CheckMap, (std::map<int, double>, bool));  // Won't compile!
};
```

Solution 1 - wrap with parentheses:

{: .good}

```
class MockFoo {
 public:
  MOCK_METHOD((std::pair<bool, int>), GetPair, ());
  MOCK_METHOD(bool, CheckMap, ((std::map<int, double>), bool));
};
```

Note that wrapping a return or argument type with parentheses is, in general, invalid C++. `MOCK_METHOD` removes the parentheses.

Solution 2 - define an alias:

{: .good}

```
class MockFoo {
 public:
  using BoolAndInt = std::pair<bool, int>;
  MOCK_METHOD(BoolAndInt, GetPair, ());
  using MapIntDouble = std::map<int, double>;
  MOCK_METHOD(bool, CheckMap, (MapIntDouble, bool));
};
```

Mocking Private or Protected Methods

You must always put a mock method definition (`MOCK_METHOD`) in a `public:` section of the mock class, regardless of the method being mocked being `public`, `protected`, or `private` in the base class. This allows `ON_CALL` and `EXPECT_CALL` to reference the mock function from outside of the mock class. (Yes, C++ allows a subclass to change the access level of a virtual function in the base class.) Example:

```
class Foo {
 public:
  ...
  virtual bool Transform(Gadget* g) = 0;

 protected:
  virtual void Resume();

 private:
  virtual int GetTimeOut();
};

class MockFoo : public Foo {
 public:
  ...
  MOCK_METHOD(bool, Transform, (Gadget* g), (override));

  // The following must be in the public section, even though the
  // methods are protected or private in the base class.
  MOCK_METHOD(void, Resume, (), (override));
  MOCK_METHOD(int, GetTimeOut, (), (override));
};
```

Mocking Overloaded Methods

You can mock overloaded functions as usual. No special attention is required:

```
class Foo {
  ...

  // Must be virtual as we'll inherit from Foo.
  virtual ~Foo();

  // Overloaded on the types and/or numbers of arguments.
  virtual int Add(Element x);
  virtual int Add(int times, Element x);

  // Overloaded on the const-ness of this object.
  virtual Bar& GetBar();
  virtual const Bar& GetBar() const;
};

class MockFoo : public Foo {
  ...
  MOCK_METHOD(int, Add, (Element x), (override));
  MOCK_METHOD(int, Add, (int times, Element x), (override));

  MOCK_METHOD(Bar&, GetBar, (), (override));
  MOCK_METHOD(const Bar&, GetBar, (), (const, override));
};
```

{: .callout .note}
Note: if you don't mock all versions of the overloaded method, the compiler will give you a warning about some methods in the base class being hidden. To fix that, use `using` to bring them in scope:

```
class MockFoo : public Foo {
  ...
  using Foo::Add;
  MOCK_METHOD(int, Add, (Element x), (override));
  // We don't want to mock int Add(int times, Element x);
  ...
};
```

Mocking Class Templates

You can mock class templates just like any class.

```
template <typename Elem>
class StackInterface {
  ...
  // Must be virtual as we'll inherit from StackInterface.
  virtual ~StackInterface();

  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};

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

Mocking Non-virtual Methods {#MockingNonVirtualMethods}

gMock can mock non-virtual functions to be used in Hi-perf dependency injection.

In this case, instead of sharing a common base class with the real class, your mock class will be *unrelated* to the real class, but contain methods with the same signatures. The syntax for mocking non-virtual methods is the *same* as mocking virtual methods (just don't add `override`):

```
// A simple packet stream class.  None of its members is virtual.
class ConcretePacketStream {
 public:
  void AppendPacket(Packet* new_packet);
  const Packet* GetPacket(size_t packet_number) const;
  size_t NumberOfPackets() const;
  ...
};

// A mock packet stream class.  It inherits from no other, but defines
// GetPacket() and NumberOfPackets().
class MockPacketStream {
 public:
  MOCK_METHOD(const Packet*, GetPacket, (size_t packet_number), (const));
  MOCK_METHOD(size_t, NumberOfPackets, (), (const));
  ...
};
```

Note that the mock class doesn't define `AppendPacket()`, unlike the real class. That's fine as long as the test doesn't need to call it.

Next, you need a way to say that you want to use `ConcretePacketStream` in production code, and use `MockPacketStream` in tests. Since the functions are not virtual and the two classes are unrelated, you must specify your choice at *compile time* (as opposed to run time).

One way to do it is to templatize your code that needs to use a packet stream. More specifically, you will give your code a template type argument for the type of the packet stream. In production, you will instantiate your template with `ConcretePacketStream` as the type argument. In tests, you will instantiate the same template with `MockPacketStream`. For example, you may write:

```
template <class PacketStream>
void CreateConnection(PacketStream* stream) { ... }

template <class PacketStream>
class PacketReader {
 public:
  void ReadPackets(PacketStream* stream, size_t packet_num);
};
```

Then you can use `CreateConnection<ConcretePacketStream>()` and `PacketReader<ConcretePacketStream>` in production code, and use `CreateConnection<MockPacketStream>()` and `PacketReader<MockPacketStream>` in tests.

```
  MockPacketStream mock_stream;
  EXPECT_CALL(mock_stream, ...)...;
  .. set more expectations on mock_stream ...
  PacketReader<MockPacketStream> reader(&mock_stream);
  ... exercise reader ...
```

Mocking Free Functions

It is not possible to directly mock a free function (i.e. a C-style function or a static method). If you need to, you can rewrite your code to use an interface (abstract class).

Instead of calling a free function (say, `OpenFile`) directly, introduce an interface for it and have a concrete subclass that calls the free function:

```
class FileInterface {
 public:
  ...
  virtual bool Open(const char* path, const char* mode) = 0;
};

class File : public FileInterface {
 public:
  ...
  bool Open(const char* path, const char* mode) override {
     return OpenFile(path, mode);
  }
};
```

Your code should talk to `FileInterface` to open a file. Now it's easy to mock out the function.

This may seem like a lot of hassle, but in practice you often have multiple related functions that you can put in the same interface, so the per-function syntactic overhead will be much lower.

If you are concerned about the performance overhead incurred by virtual functions, and profiling confirms your concern, you can combine this with the recipe for mocking non-virtual methods.

Old-Style `MOCK_METHODn` Macros

Before the generic `MOCK_METHOD` macro was introduced in 2018, mocks where created using a family of macros collectively called `MOCK_METHODn`. These macros are still supported, though migration to the new `MOCK_METHOD` is recommended.

The macros in the `MOCK_METHODn` family differ from `MOCK_METHOD`:

- The general structure is `MOCK_METHODn(MethodName, ReturnType(Args))`, instead of `MOCK_METHOD(ReturnType, MethodName, (Args))`.
- The number `n` must equal the number of arguments.
- When mocking a const method, one must use `MOCK_CONST_METHODn`.
- When mocking a class template, the macro name must be suffixed with `_T`.
- In order to specify the call type, the macro name must be suffixed with `_WITH_CALLTYPE`, and the call type is the first macro argument.

Old macros and their new equivalents:

Simple	
Old	`MOCK_METHOD1(Foo, bool(int))`
New	`MOCK_METHOD(bool, Foo, (int))`

Const Method

Old
`MOCK_CONST_METHOD1(Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int), (const))`

Method in a Class Template

Old
`MOCK_METHOD1_T(Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int))`

Const Method in a Class Template

Old
`MOCK_CONST_METHOD1_T(Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int), (const))`

Method with Call Type

Old
`MOCK_METHOD1_WITH_CALLTYPE(STDMETHODCALLTYPE, Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int), (Calltype(STDMETHODCALLTYPE)))`

Const Method with Call Type

Old
`MOCK_CONST_METHOD1_WITH_CALLTYPE(STDMETHODCALLTYPE, Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int), (const, Calltype(STDMETHODCALLTYPE)))`

Method with Call Type in a Class Template

Old
`MOCK_METHOD1_T_WITH_CALLTYPE(STDMETHODCALLTYPE, Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int), (Calltype(STDMETHODCALLTYPE)))`

Const Method with Call Type in a Class Template

Old
`MOCK_CONST_METHOD1_T_WITH_CALLTYPE(STDMETHODCALLTYPE, Foo, bool(int))`

New
`MOCK_METHOD(bool, Foo, (int), (const, Calltype(STDMETHODCALLTYPE)))`

The Nice, the Strict, and the Naggy {#NiceStrictNaggy}

If a mock method has no `EXPECT_CALL` spec but is called, we say that it's an "uninteresting call", and the default action (which can be specified using `ON_CALL()`) of the method will be taken. Currently, an uninteresting call will also by default cause gMock to print a warning.

However, sometimes you may want to ignore these uninteresting calls, and sometimes you may want to treat them as errors. gMock lets you make the decision on a per-mock-object basis.

Suppose your test uses a mock class `MockFoo`:

```
TEST(...) {
  MockFoo mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

If a method of `mock_foo` other than `DoThis()` is called, you will get a warning. However, if you rewrite your test to use `NiceMock<MockFoo>` instead, you can suppress the warning:

```
using ::testing::NiceMock;

TEST(...) {
  NiceMock<MockFoo> mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

`NiceMock<MockFoo>` is a subclass of `MockFoo`, so it can be used wherever `MockFoo` is accepted.

It also works if `MockFoo`'s constructor takes some arguments, as `NiceMock<MockFoo>` "inherits" `MockFoo`'s constructors:

```
using ::testing::NiceMock;

TEST(...) {
  NiceMock<MockFoo> mock_foo(5, "hi");  // Calls MockFoo(5, "hi").
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

The usage of `StrictMock` is similar, except that it makes all uninteresting calls failures:

```
using ::testing::StrictMock;

TEST(...) {
  StrictMock<MockFoo> mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...

  // The test will fail if a method of mock_foo other than DoThis()
  // is called.
}
```

{: .callout .note}
NOTE: `NiceMock` and `StrictMock` only affects *uninteresting* calls (calls of *methods* with no expectations); they do not affect *unexpected* calls (calls of methods with expectations, but they don't match). See [Understanding Uninteresting vs Unexpected Calls](#).

There are some caveats though (sadly they are side effects of C++'s limitations):

1. `NiceMock<MockFoo>` and `StrictMock<MockFoo>` only work for mock methods defined using the `MOCK_METHOD` macro **directly** in the `MockFoo` class. If a mock method is defined in a **base class** of `MockFoo`, the "nice" or "strict" modifier may not affect it, depending on the compiler. In particular, nesting `NiceMock` and `StrictMock` (e.g. `NiceMock<StrictMock<MockFoo> >`) is **not** supported.
2. `NiceMock<MockFoo>` and `StrictMock<MockFoo>` may not work correctly if the destructor of `MockFoo` is not virtual. We would like to fix this, but it requires cleaning up existing tests.

Finally, you should be **very cautious** about when to use naggy or strict mocks, as they tend to make tests more brittle and harder to maintain. When you refactor your code without changing its externally visible behavior, ideally you shouldn't need to update any tests. If your code interacts with a naggy mock, however, you may start to get spammed with warnings as the result of your change. Worse, if your code interacts with a strict mock, your tests may start to fail and you'll be forced to fix them. Our general recommendation is to use nice mocks (not yet the default) most of the time, use naggy mocks (the current default) when developing or debugging tests, and use strict mocks only as the last resort.

Simplifying the Interface without Breaking Existing Code {#SimplerInterfaces}

Sometimes a method has a long list of arguments that is mostly uninteresting. For example:

```
class LogSink {
 public:
  ...
  virtual void send(LogSeverity severity, const char* full_filename,
                    const char* base_filename, int line,
                    const struct tm* tm_time,
                    const char* message, size_t message_len) = 0;
};
```

This method's argument list is lengthy and hard to work with (the `message` argument is not even 0-terminated). If we mock it as is, using the mock will be awkward. If, however, we try to simplify this interface, we'll need to fix all clients depending on it, which is often infeasible.

The trick is to redispatch the method in the mock class:

```
class ScopedMockLog : public LogSink {
 public:
  ...
  void send(LogSeverity severity, const char* full_filename,
                    const char* base_filename, int line, const tm* tm_time,
                    const char* message, size_t message_len) override {
    // We are only interested in the log severity, full file name, and
    // log message.
    Log(severity, full_filename, std::string(message, message_len));
  }

  // Implements the mock method:
  //
  //   void Log(LogSeverity severity,
  //            const string& file_path,
  //            const string& message);
  MOCK_METHOD(void, Log,
              (LogSeverity severity, const string& file_path,
               const string& message));
};
```

By defining a new mock method with a trimmed argument list, we make the mock class more user-friendly.

This technique may also be applied to make overloaded methods more amenable to mocking. For example, when overloads have been used to implement default arguments:

```
class MockTurtleFactory : public TurtleFactory {
 public:
  Turtle* MakeTurtle(int length, int weight) override { ... }
  Turtle* MakeTurtle(int length, int weight, int speed) override { ... }

  // the above methods delegate to this one:
  MOCK_METHOD(Turtle*, DoMakeTurtle, ());
};
```

This allows tests that don't care which overload was invoked to avoid specifying argument matchers:

```
ON_CALL(factory, DoMakeTurtle)
    .WillByDefault(Return(MakeMockTurtle()));
```

Alternative to Mocking Concrete Classes

Often you may find yourself using classes that don't implement interfaces. In order to test your code that uses such a class (let's call it `Concrete`), you may be tempted to make the methods of `Concrete` virtual and then mock it.

Try not to do that.

Making a non-virtual function virtual is a big decision. It creates an extension point where subclasses can tweak your class' behavior. This weakens your control on the class because now it's harder to maintain the class invariants. You should make a function virtual only when there is a valid reason for a subclass to override it.

Mocking concrete classes directly is problematic as it creates a tight coupling between the class and the tests - any small change in the class may invalidate your tests and make test maintenance a pain.

To avoid such problems, many programmers have been practicing "coding to interfaces": instead of talking to the `Concrete` class, your code would define an interface and talk to it. Then you implement that interface as an adaptor on top of `Concrete`. In tests, you can easily mock that interface to observe how your code is doing.

This technique incurs some overhead:

- You pay the cost of virtual function calls (usually not a problem).
- There is more abstraction for the programmers to learn.

However, it can also bring significant benefits in addition to better testability:

- `Concrete`'s API may not fit your problem domain very well, as you may not be the only client it tries to serve. By designing your own interface, you have a chance to tailor it to your need - you may add higher-level

functionalities, rename stuff, etc instead of just trimming the class. This allows you to write your code (user of the interface) in a more natural way, which means it will be more readable, more maintainable, and you'll be more productive.
- If `Concrete`'s implementation ever has to change, you don't have to rewrite everywhere it is used. Instead, you can absorb the change in your implementation of the interface, and your other code and tests will be insulated from this change.

Some people worry that if everyone is practicing this technique, they will end up writing lots of redundant code. This concern is totally understandable. However, there are two reasons why it may not be the case:

- Different projects may need to use `Concrete` in different ways, so the best interfaces for them will be different. Therefore, each of them will have its own domain-specific interface on top of `Concrete`, and they will not be the same code.
- If enough projects want to use the same interface, they can always share it, just like they have been sharing `Concrete`. You can check in the interface and the adaptor somewhere near `Concrete` (perhaps in a `contrib` sub-directory) and let many projects use it.

You need to weigh the pros and cons carefully for your particular problem, but I'd like to assure you that the Java community has been practicing this for a long time and it's a proven effective technique applicable in a wide variety of situations. :-)

Delegating Calls to a Fake {#DelegatingToFake}

Some times you have a non-trivial fake implementation of an interface. For example:

```
class Foo {
 public:
  virtual ~Foo() {}
  virtual char DoThis(int n) = 0;
  virtual void DoThat(const char* s, int* p) = 0;
};

class FakeFoo : public Foo {
 public:
  char DoThis(int n) override {
    return (n > 0) ? '+' :
           (n < 0) ? '-' : '0';
  }

  void DoThat(const char* s, int* p) override {
    *p = strlen(s);
  }
};
```

Now you want to mock this interface such that you can set expectations on it. However, you also want to use `FakeFoo` for the default behavior, as duplicating it in the mock object is, well, a lot of work.

When you define the mock class using gMock, you can have it delegate its default action to a fake class you already have, using this pattern:

```
class MockFoo : public Foo {
 public:
  // Normal mock method definitions using gMock.
  MOCK_METHOD(char, DoThis, (int n), (override));
  MOCK_METHOD(void, DoThat, (const char* s, int* p), (override));

  // Delegates the default actions of the methods to a FakeFoo object.
  // This must be called *before* the custom ON_CALL() statements.
  void DelegateToFake() {
    ON_CALL(*this, DoThis).WillByDefault([this](int n) {
      return fake_.DoThis(n);
    });
    ON_CALL(*this, DoThat).WillByDefault([this](const char* s, int* p) {
      fake_.DoThat(s, p);
    });
  }

 private:
  FakeFoo fake_;  // Keeps an instance of the fake in the mock.
};
```

With that, you can use `MockFoo` in your tests as usual. Just remember that if you don't explicitly set an action in an `ON_CALL()` or `EXPECT_CALL()`, the fake will be called upon to do it.:

```
using ::testing::_;

TEST(AbcTest, Xyz) {
  MockFoo foo;

  foo.DelegateToFake();  // Enables the fake for delegation.

  // Put your ON_CALL(foo, ...)s here, if any.

  // No action specified, meaning to use the default action.
  EXPECT_CALL(foo, DoThis(5));
  EXPECT_CALL(foo, DoThat(_, _));

  int n = 0;
  EXPECT_EQ('+', foo.DoThis(5));  // FakeFoo::DoThis() is invoked.
  foo.DoThat("Hi", &n);  // FakeFoo::DoThat() is invoked.
  EXPECT_EQ(2, n);
}
```

Some tips:

- If you want, you can still override the default action by providing your own `ON_CALL()` or using `.WillOnce()` / `.WillRepeatedly()` in `EXPECT_CALL()`.
- In `DelegateToFake()`, you only need to delegate the methods whose fake implementation you intend to use.
- The general technique discussed here works for overloaded methods, but you'll need to tell the compiler which version you mean. To disambiguate a mock function (the one you specify inside the parentheses of `ON_CALL()`), use this technique; to disambiguate a fake function (the one you place inside `Invoke()`), use a `static_cast` to specify the function's type. For instance, if class `Foo` has methods `char DoThis(int n)` and `bool DoThis(double x) const`, and you want to invoke the latter, you need to write `Invoke(&fake_, static_cast<bool (FakeFoo::*)(double) const>(&FakeFoo::DoThis))` instead of `Invoke(&fake_, &FakeFoo::DoThis)` (The strange-looking thing inside the angled brackets of `static_cast` is the type of a function pointer to the second `DoThis()` method.).
- Having to mix a mock and a fake is often a sign of something gone wrong. Perhaps you haven't got used to the interaction-based way of testing yet. Or perhaps your interface is taking on too many roles and should be split up. Therefore, **don't abuse this**. We would only recommend to do it as an intermediate step when you are refactoring your code.

Regarding the tip on mixing a mock and a fake, here's an example on why it may be a bad sign: Suppose you have a class `System` for low-level system operations. In particular, it does file and I/O operations. And suppose you want to test how your code uses `System` to do I/O, and you just want the file operations to work normally. If you mock out the entire `System` class, you'll have to provide a fake implementation for the file operation part, which suggests that `System` is taking on too many roles.

Instead, you can define a `FileOps` interface and an `IOOps` interface and split `System`'s functionalities into the two. Then you can mock `IOOps` without mocking `FileOps`.

Delegating Calls to a Real Object

When using testing doubles (mocks, fakes, stubs, and etc), sometimes their behaviors will differ from those of the real objects. This difference could be either intentional (as in simulating an error such that you can test the error handling code) or unintentional. If your mocks have different behaviors than the real objects by mistake, you could end up with code that passes the tests but fails in production.

You can use the *delegating-to-real* technique to ensure that your mock has the same behavior as the real object while retaining the ability to validate calls. This technique is very similar to the delegating-to-fake technique, the difference being that we use a real object instead of a fake. Here's an example:

```
using ::testing::AtLeast;

class MockFoo : public Foo {
 public:
  MockFoo() {
    // By default, all calls are delegated to the real object.
    ON_CALL(*this, DoThis).WillByDefault([this](int n) {
      return real_.DoThis(n);
    });
    ON_CALL(*this, DoThat).WillByDefault([this](const char* s, int* p) {
      real_.DoThat(s, p);
    });
    ...
  }
  MOCK_METHOD(char, DoThis, ...);
  MOCK_METHOD(void, DoThat, ...);
  ...
 private:
  Foo real_;
};

...
  MockFoo mock;
  EXPECT_CALL(mock, DoThis())
      .Times(3);
  EXPECT_CALL(mock, DoThat("Hi"))
      .Times(AtLeast(1));
  ... use mock in test ...
```

With this, gMock will verify that your code made the right calls (with the right arguments, in the right order, called the right number of times, etc), and a real object will answer the calls (so the behavior will be the same as in production). This gives you the best of both worlds.

Delegating Calls to a Parent Class

Ideally, you should code to interfaces, whose methods are all pure virtual. In reality, sometimes you do need to mock a virtual method that is not pure (i.e, it already has an implementation). For example:

```
class Foo {
 public:
  virtual ~Foo();

  virtual void Pure(int n) = 0;
  virtual int Concrete(const char* str) { ... }
};

class MockFoo : public Foo {
 public:
  // Mocking a pure method.
  MOCK_METHOD(void, Pure, (int n), (override));
  // Mocking a concrete method.  Foo::Concrete() is shadowed.
  MOCK_METHOD(int, Concrete, (const char* str), (override));
};
```

Sometimes you may want to call `Foo::Concrete()` instead of `MockFoo::Concrete()`. Perhaps you want to do it as part of a stub action, or perhaps your test doesn't need to mock `Concrete()` at all (but it would be oh-so painful to have to define a new mock class whenever you don't need to mock one of its methods).

You can call `Foo::Concrete()` inside an action by:

```
...
  EXPECT_CALL(foo, Concrete).WillOnce([&foo](const char* str) {
    return foo.Foo::Concrete(str);
  });
```

or tell the mock object that you don't want to mock `Concrete()`:

```
...
  ON_CALL(foo, Concrete).WillByDefault([&foo](const char* str) {
    return foo.Foo::Concrete(str);
  });
```

(Why don't we just write `{ return foo.Concrete(str); }`? If you do that, `MockFoo::Concrete()` will be called (and cause an infinite recursion) since `Foo::Concrete()` is virtual. That's just how C++ works.)

Using Matchers

Matching Argument Values Exactly

You can specify exactly which arguments a mock method is expecting:

```
using ::testing::Return;
...
  EXPECT_CALL(foo, DoThis(5))
      .WillOnce(Return('a'));
  EXPECT_CALL(foo, DoThat("Hello", bar));
```

Using Simple Matchers

You can use matchers to match arguments that have a certain property:

```
using ::testing::NotNull;
using ::testing::Return;
...
  EXPECT_CALL(foo, DoThis(Ge(5)))  // The argument must be >= 5.
      .WillOnce(Return('a'));
  EXPECT_CALL(foo, DoThat("Hello", NotNull()));
      // The second argument must not be NULL.
```

A frequently used matcher is `_`, which matches anything:

```
  EXPECT_CALL(foo, DoThat(_, NotNull()));
```

Combining Matchers {#CombiningMatchers}

You can build complex matchers from existing ones using `AllOf()`, `AllOfArray()`, `AnyOf()`, `AnyOfArray()` and `Not()`:

```
using ::testing::AllOf;
using ::testing::Gt;
using ::testing::HasSubstr;
using ::testing::Ne;
using ::testing::Not;
...
  // The argument must be > 5 and != 10.
  EXPECT_CALL(foo, DoThis(AllOf(Gt(5),
                                Ne(10))));

  // The first argument must not contain sub-string "blah".
  EXPECT_CALL(foo, DoThat(Not(HasSubstr("blah")),
                          NULL));
```

Matchers are function objects, and parametrized matchers can be composed just like any other function. However because their types can be long and rarely provide meaningful information, it can be easier to express them with C++14 generic lambdas to avoid specifying types. For example,

```
using ::testing::Contains;
using ::testing::Property;

inline constexpr auto HasFoo = [](const auto& f) {
  return Property(&MyClass::foo, Contains(f));
};
...
  EXPECT_THAT(x, HasFoo("blah"));
```

Casting Matchers {#SafeMatcherCast}

gMock matchers are statically typed, meaning that the compiler can catch your mistake if you use a matcher of the wrong type (for example, if you use `Eq(5)` to match a `string` argument). Good for you!

Sometimes, however, you know what you're doing and want the compiler to give you some slack. One example is that you have a matcher for `long` and the argument you want to match is `int`. While the two types aren't exactly the same, there is nothing really wrong with using a `Matcher<long>` to match an `int` - after all, we can first convert the `int` argument to a `long` losslessly before giving it to the matcher.

To support this need, gMock gives you the `SafeMatcherCast<T>(m)` function. It casts a matcher `m` to type `Matcher<T>`. To ensure safety, gMock checks that (let `U` be the type `m` accepts :

1. Type `T` can be *implicitly* cast to type `U`;

2. When both `T` and `U` are built-in arithmetic types (`bool`, integers, and floating-point numbers), the conversion from `T` to `U` is not lossy (in other words, any value representable by `T` can also be represented by `U`); and
3. When `U` is a reference, `T` must also be a reference (as the underlying matcher may be interested in the address of the `U` value).

The code won't compile if any of these conditions isn't met.

Here's one example:

```
using ::testing::SafeMatcherCast;

// A base class and a child class.
class Base { ... };
class Derived : public Base { ... };

class MockFoo : public Foo {
 public:
  MOCK_METHOD(void, DoThis, (Derived* derived), (override));
};

...
  MockFoo foo;
  // m is a Matcher<Base*> we got from somewhere.
  EXPECT_CALL(foo, DoThis(SafeMatcherCast<Derived*>(m)));
```

If you find `SafeMatcherCast<T>(m)` too limiting, you can use a similar function `MatcherCast<T>(m)`. The difference is that `MatcherCast` works as long as you can `static_cast` type `T` to type `U`.

`MatcherCast` essentially lets you bypass C++'s type system (`static_cast` isn't always safe as it could throw away information, for example), so be careful not to misuse/abuse it.

Selecting Between Overloaded Functions {#SelectOverload}

If you expect an overloaded function to be called, the compiler may need some help on which overloaded version it is.

To disambiguate functions overloaded on the const-ness of this object, use the `Const()` argument wrapper.

```
using ::testing::ReturnRef;

class MockFoo : public Foo {
  ...
  MOCK_METHOD(Bar&, GetBar, (), (override));
  MOCK_METHOD(const Bar&, GetBar, (), (const, override));
};

...
  MockFoo foo;
  Bar bar1, bar2;
  EXPECT_CALL(foo, GetBar())         // The non-const GetBar().
      .WillOnce(ReturnRef(bar1));
  EXPECT_CALL(Const(foo), GetBar())  // The const GetBar().
      .WillOnce(ReturnRef(bar2));
```

(`Const()` is defined by gMock and returns a `const` reference to its argument.)

To disambiguate overloaded functions with the same number of arguments but different argument types, you may need to specify the exact type of a matcher, either by wrapping your matcher in `Matcher<type>()`, or using a matcher whose type is fixed (`TypedEq<type>`, `An<type>()`, etc):

```
using ::testing::An;
using ::testing::Matcher;
using ::testing::TypedEq;

class MockPrinter : public Printer {
 public:
  MOCK_METHOD(void, Print, (int n), (override));
  MOCK_METHOD(void, Print, (char c), (override));
};

TEST(PrinterTest, Print) {
  MockPrinter printer;

  EXPECT_CALL(printer, Print(An<int>()));            // void Print(int);
  EXPECT_CALL(printer, Print(Matcher<int>(Lt(5))));  // void Print(int);
  EXPECT_CALL(printer, Print(TypedEq<char>('a')));   // void Print(char);

  printer.Print(3);
  printer.Print(6);
  printer.Print('a');
}
```

Performing Different Actions Based on the Arguments

When a mock method is called, the *last* matching expectation that's still active will be selected (think "newer overrides older"). So, you can make a method do different things depending on its argument values like this:

```
using ::testing::_;
using ::testing::Lt;
using ::testing::Return;
...
  // The default case.
  EXPECT_CALL(foo, DoThis(_))
      .WillRepeatedly(Return('b'));
  // The more specific case.
  EXPECT_CALL(foo, DoThis(Lt(5)))
      .WillRepeatedly(Return('a'));
```

Now, if `foo.DoThis()` is called with a value less than 5, `'a'` will be returned; otherwise `'b'` will be returned.

Matching Multiple Arguments as a Whole

Sometimes it's not enough to match the arguments individually. For example, we may want to say that the first argument must be less than the second argument. The `With()` clause allows us to match all arguments of a mock function as a whole. For example,

```
using ::testing::_;
using ::testing::Ne;
using ::testing::Lt;
...
  EXPECT_CALL(foo, InRange(Ne(0), _))
      .With(Lt());
```

says that the first argument of `InRange()` must not be 0, and must be less than the second argument.

The expression inside `With()` must be a matcher of type `Matcher<std::tuple<A1, ..., An>>`, where `A1`, ..., `An` are the types of the function arguments.

You can also write `AllArgs(m)` instead of `m` inside `.With()`. The two forms are equivalent, but `.With(AllArgs(Lt()))` is more readable than `.With(Lt())`.

You can use `Args<k1, ..., kn>(m)` to match the `n` selected arguments (as a tuple) against `m`. For example,

```
using ::testing::_;
using ::testing::AllOf;
using ::testing::Args;
using ::testing::Lt;
...
  EXPECT_CALL(foo, Blah)
      .With(AllOf(Args<0, 1>(Lt()), Args<1, 2>(Lt())));
```

says that `Blah` will be called with arguments `x`, `y`, and `z` where `x < y < z`. Note that in this example, it wasn't necessary to specify the positional matchers.

As a convenience and example, gMock provides some matchers for 2-tuples, including the `Lt()` matcher above. See Multi-argument Matchers for the complete list.

Note that if you want to pass the arguments to a predicate of your own (e.g. `.With(Args<0, 1>(Truly(&MyPredicate)))`), that predicate MUST be written to take a `std::tuple` as its argument; gMock will pass the `n` selected arguments as *one* single tuple to the predicate.

Using Matchers as Predicates

Have you noticed that a matcher is just a fancy predicate that also knows how to describe itself? Many existing algorithms take predicates as arguments (e.g. those defined in STL's `<algorithm>` header), and it would be a shame if gMock matchers were not allowed to participate.

Luckily, you can use a matcher where a unary predicate functor is expected by wrapping it inside the `Matches()` function. For example,

```
#include <algorithm>
#include <vector>

using ::testing::Matches;
using ::testing::Ge;

vector<int> v;
...
// How many elements in v are >= 10?
const int count = count_if(v.begin(), v.end(), Matches(Ge(10)));
```

Since you can build complex matchers from simpler ones easily using gMock, this gives you a way to conveniently construct composite predicates (doing the same using STL's `<functional>` header is just painful). For example, here's a predicate that's satisfied by any number that is >= 0, <= 100, and != 50:

```
using testing::AllOf;
using testing::Ge;
using testing::Le;
using testing::Matches;
using testing::Ne;
...
Matches(AllOf(Ge(0), Le(100), Ne(50)))
```

Using Matchers in googletest Assertions

See `EXPECT_THAT` in the Assertions Reference.

Using Predicates as Matchers

gMock provides a set of built-in matchers for matching arguments with expected values—see the Matchers Reference for more information.
In case you find the built-in set lacking, you can use an arbitrary unary predicate function or functor as a matcher - as long as the predicate accepts a value of the type you want. You do this by wrapping the predicate inside the `Truly()` function, for example:

```
using ::testing::Truly;

int IsEven(int n) { return (n % 2) == 0 ? 1 : 0; }
...
  // Bar() must be called with an even number.
  EXPECT_CALL(foo, Bar(Truly(IsEven)));
```

Note that the predicate function / functor doesn't have to return `bool`. It works as long as the return value can be used as the condition in in statement `if (condition) ...`.

Matching Arguments that Are Not Copyable

When you do an `EXPECT_CALL(mock_obj, Foo(bar))`, gMock saves away a copy of `bar`. When `Foo()` is called later, gMock compares the argument to `Foo()` with the saved copy of `bar`. This way, you don't need to worry about `bar` being modified or destroyed after the `EXPECT_CALL()` is executed. The same is true when you use matchers like `Eq(bar)`, `Le(bar)`, and so on.

But what if `bar` cannot be copied (i.e. has no copy constructor)? You could define your own matcher function or callback and use it with `Truly()`, as the previous couple of recipes have shown. Or, you may be able to get away from it if you can guarantee that `bar` won't be changed after the `EXPECT_CALL()` is executed. Just tell gMock that it should save a reference to `bar`, instead of a copy of it. Here's how:

```
using ::testing::Eq;
using ::testing::Lt;
...
  // Expects that Foo()'s argument == bar.
  EXPECT_CALL(mock_obj, Foo(Eq(std::ref(bar))));

  // Expects that Foo()'s argument < bar.
  EXPECT_CALL(mock_obj, Foo(Lt(std::ref(bar))));
```

Remember: if you do this, don't change `bar` after the `EXPECT_CALL()`, or the result is undefined.

Validating a Member of an Object

Often a mock function takes a reference to object as an argument. When matching the argument, you may not want to compare the entire object against a fixed object, as that may be over-specification. Instead, you may need to validate a certain member variable or the result of a certain getter method of the object. You can do this with `Field()` and `Property()`. More specifically,

```
Field(&Foo::bar, m)
```

is a matcher that matches a `Foo` object whose `bar` member variable satisfies matcher `m`.

```
Property(&Foo::baz, m)
```

is a matcher that matches a `Foo` object whose `baz()` method returns a value that satisfies matcher `m`.

For example:

Expression	Description
`Field(&Foo::number, Ge(3))`	Matches `x` where `x.number >= 3`.
`Property(&Foo::name, StartsWith("John "))`	Matches `x` where `x.name()` starts with `"John "`.

Note that in `Property(&Foo::baz, ...)`, method `baz()` must take no argument and be declared as `const`. Don't use `Property()` against member functions that you do not own, because taking addresses of functions is fragile and generally not part of the contract of the function.

`Field()` and `Property()` can also match plain pointers to objects. For instance,

```
using ::testing::Field;
using ::testing::Ge;
...
Field(&Foo::number, Ge(3))
```

matches a plain pointer `p` where `p->number >= 3`. If `p` is `NULL`, the match will always fail regardless of the inner matcher.

What if you want to validate more than one members at the same time? Remember that there are `AllOf()` and `AllOfArray()`.

Finally `Field()` and `Property()` provide overloads that take the field or property names as the first argument to include it in the error message. This can be useful when creating combined matchers.

```
using ::testing::AllOf;
using ::testing::Field;
using ::testing::Matcher;
using ::testing::SafeMatcherCast;

Matcher<Foo> IsFoo(const Foo& foo) {
  return AllOf(Field("some_field", &Foo::some_field, foo.some_field),
               Field("other_field", &Foo::other_field, foo.other_field),
               Field("last_field", &Foo::last_field, foo.last_field));
}
```

Validating the Value Pointed to by a Pointer Argument

C++ functions often take pointers as arguments. You can use matchers like `IsNull()`, `NotNull()`, and other comparison matchers to match a pointer, but what if you want to make sure the value *pointed to* by the pointer, instead of the pointer itself, has a certain property? Well, you can use the `Pointee(m)` matcher.

`Pointee(m)` matches a pointer if and only if `m` matches the value the pointer points to. For example:

```
using ::testing::Ge;
using ::testing::Pointee;
...
  EXPECT_CALL(foo, Bar(Pointee(Ge(3))));
```

expects `foo.Bar()` to be called with a pointer that points to a value greater than or equal to 3.

One nice thing about `Pointee()` is that it treats a `NULL` pointer as a match failure, so you can write `Pointee(m)` instead of

```
using ::testing::AllOf;
using ::testing::NotNull;
using ::testing::Pointee;
...
  AllOf(NotNull(), Pointee(m))
```

without worrying that a `NULL` pointer will crash your test.

Also, did we tell you that `Pointee()` works with both raw pointers **and** smart pointers (`std::unique_ptr`, `std::shared_ptr`, etc)?

What if you have a pointer to pointer? You guessed it - you can use nested `Pointee()` to probe deeper inside the value. For example, `Pointee(Pointee(Lt(3)))` matches a pointer that points to a pointer that points to a number less than 3 (what a mouthful...).

Defining a Custom Matcher Class {#CustomMatcherClass}

Most matchers can be simply defined using the MATCHER* macros, which are terse and flexible, and produce good error messages. However, these macros are not very explicit about the interfaces they create and are not always suitable, especially for matchers that will be widely reused.

For more advanced cases, you may need to define your own matcher class. A custom matcher allows you to test a specific invariant property of that object. Let's take a look at how to do so.

Imagine you have a mock function that takes an object of type `Foo`, which has an `int bar()` method and an `int baz()` method. You want to constrain that the argument's `bar()` value plus its `baz()` value is a given number. (This is an invariant.) Here's how we can write and use a matcher class to do so:

```
class BarPlusBazEqMatcher {
 public:
  using is_gtest_matcher = void;

  explicit BarPlusBazEqMatcher(int expected_sum)
      : expected_sum_(expected_sum) {}

  bool MatchAndExplain(const Foo& foo,
                       std::ostream* /* listener */) const {
    return (foo.bar() + foo.baz()) == expected_sum_;
  }

  void DescribeTo(std::ostream* os) const {
    *os << "bar() + baz() equals " << expected_sum_;
  }

  void DescribeNegationTo(std::ostream* os) const {
    *os << "bar() + baz() does not equal " << expected_sum_;
  }
 private:
  const int expected_sum_;
};

::testing::Matcher<const Foo&> BarPlusBazEq(int expected_sum) {
  return BarPlusBazEqMatcher(expected_sum);
}

...
  Foo foo;
  EXPECT_CALL(foo, BarPlusBazEq(5))...;
```

Matching Containers

Sometimes an STL container (e.g. list, vector, map, ...) is passed to a mock function and you may want to validate it. Since most STL containers support the `==` operator, you can write `Eq(expected_container)` or simply `expected_container` to match a container exactly.

Sometimes, though, you may want to be more flexible (for example, the first element must be an exact match, but the second element can be any positive number, and so on). Also, containers used in tests often have a small number of elements, and having to define the expected container out-of-line is a bit of a hassle.

You can use the `ElementsAre()` or `UnorderedElementsAre()` matcher in such cases:

```
using ::testing::_;
using ::testing::ElementsAre;
using ::testing::Gt;
...
  MOCK_METHOD(void, Foo, (const vector<int>& numbers), (override));
...
  EXPECT_CALL(mock, Foo(ElementsAre(1, Gt(0), _, 5)));
```

The above matcher says that the container must have 4 elements, which must be 1, greater than 0, anything, and 5 respectively.

If you instead write:

```
using ::testing::_;
using ::testing::Gt;
using ::testing::UnorderedElementsAre;
...
  MOCK_METHOD(void, Foo, (const vector<int>& numbers), (override));
...
  EXPECT_CALL(mock, Foo(UnorderedElementsAre(1, Gt(0), _, 5)));
```

It means that the container must have 4 elements, which (under some permutation) must be 1, greater than 0, anything, and 5 respectively.

As an alternative you can place the arguments in a C-style array and use `ElementsAreArray()` or `UnorderedElementsAreArray()` instead:

```
using ::testing::ElementsAreArray;
...
  // ElementsAreArray accepts an array of element values.
  const int expected_vector1[] = {1, 5, 2, 4, ...};
  EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector1)));

  // Or, an array of element matchers.
  Matcher<int> expected_vector2[] = {1, Gt(2), _, 3, ...};
  EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector2)));
```

In case the array needs to be dynamically created (and therefore the array size cannot be inferred by the compiler), you can give `ElementsAreArray()` an additional argument to specify the array size:

```
using ::testing::ElementsAreArray;
...
  int* const expected_vector3 = new int[count];
  ... fill expected_vector3 with values ...
  EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector3, count)));
```

Use `Pair` when comparing maps or other associative containers.

{% raw %}

```
using ::testing::UnorderedElementsAre;
using ::testing::Pair;
...
  absl::flat_hash_map<string, int> m = {{"a", 1}, {"b", 2}, {"c", 3}};
  EXPECT_THAT(m, UnorderedElementsAre(
      Pair("a", 1), Pair("b", 2), Pair("c", 3)));
```

{% endraw %}

Tips:

- `ElementsAre*()` can be used to match *any* container that implements the STL iterator pattern (i.e. it has a `const_iterator` type and supports `begin()/end()`), not just the ones defined in STL. It will even work with container types yet to be written - as long as they follows the above pattern.
- You can use nested `ElementsAre*()` to match nested (multi-dimensional) containers.
- If the container is passed by pointer instead of by reference, just write `Pointee(ElementsAre*(...))`.
- The order of elements *matters* for `ElementsAre*()`. If you are using it with containers whose element order are undefined (such as a `std::unordered_map`) you should use `UnorderedElementsAre`.

Sharing Matchers

Under the hood, a gMock matcher object consists of a pointer to a ref-counted implementation object. Copying matchers is allowed and very efficient, as only the pointer is copied. When the last matcher that references the implementation object dies, the implementation object will be deleted.

Therefore, if you have some complex matcher that you want to use again and again, there is no need to build it every time. Just assign it to a matcher variable and use that variable repeatedly! For example,

```
using ::testing::AllOf;
using ::testing::Gt;
using ::testing::Le;
using ::testing::Matcher;
...
  Matcher<int> in_range = AllOf(Gt(5), Le(10));
  ... use in_range as a matcher in multiple EXPECT_CALLs ...
```

Matchers must have no side-effects {#PureMatchers}

{: .callout .warning}
WARNING: gMock does not guarantee when or how many times a matcher will be invoked. Therefore, all matchers must be *purely functional*: they cannot have any side effects, and the match result must not depend on anything other than the matcher's parameters and the value being matched.

This requirement must be satisfied no matter how a matcher is defined (e.g., if it is one of the standard matchers, or a custom matcher). In particular, a matcher can never call a mock function, as that will affect the state of the mock object and gMock.

Setting Expectations

Knowing When to Expect {#UseOnCall}

`ON_CALL` is likely the *single most under-utilized construct* in gMock.

There are basically two constructs for defining the behavior of a mock object: `ON_CALL` and `EXPECT_CALL`. The difference? `ON_CALL` defines what happens when a mock method is called, but *doesn't imply any expectation on the method being called*. `EXPECT_CALL` not only defines the behavior, but also sets an expectation that *the method will be called with the given arguments, for the given number of times* (and *in the given order* when you specify the order too).

Since `EXPECT_CALL` does more, isn't it better than `ON_CALL`? Not really. Every `EXPECT_CALL` adds a constraint on the behavior of the code under test. Having more constraints than necessary is *baaad* - even worse than not having enough constraints.

This may be counter-intuitive. How could tests that verify more be worse than tests that verify less? Isn't verification the whole point of tests?

The answer lies in *what* a test should verify. **A good test verifies the contract of the code.** If a test over-specifies, it doesn't leave enough freedom to the implementation. As a result, changing the implementation without breaking the contract (e.g. refactoring and optimization), which should be perfectly fine to do, can break such tests. Then you have to spend time fixing them, only to see them broken again the next time the implementation is changed.

Keep in mind that one doesn't have to verify more than one property in one test. In fact, **it's a good style to verify only one thing in one test.** If you do that, a bug will likely break only one or two tests instead of dozens (which case would you rather debug?). If you are also in the habit of giving tests descriptive names that tell what they verify, you can often easily guess what's wrong just from the test log itself.

So use `ON_CALL` by default, and only use `EXPECT_CALL` when you actually intend to verify that the call is made. For example, you may have a bunch of `ON_CALL`s in your test fixture to set the common mock behavior shared by all tests in the same group, and write (scarcely) different `EXPECT_CALL`s in different `TEST_F`s to verify different aspects of the code's behavior. Compared with the style where each `TEST` has many `EXPECT_CALL`s, this leads to tests that are more resilient to implementational changes (and thus less likely to require maintenance) and makes the intent of the tests more obvious (so they are easier to maintain when you do need to maintain them).

If you are bothered by the "Uninteresting mock function call" message printed when a mock method without an `EXPECT_CALL` is called, you may use a `NiceMock` instead to suppress all such messages for the mock object, or suppress the message for specific methods by adding `EXPECT_CALL(...).Times(AnyNumber())`. DO NOT suppress it by blindly adding an `EXPECT_CALL(...)`, or you'll have a test that's a pain to maintain.

Ignoring Uninteresting Calls

If you are not interested in how a mock method is called, just don't say anything about it. In this case, if the method is ever called, gMock will perform its default action to allow the test program to continue. If you are not happy with the default action taken by gMock, you can override it using `DefaultValue<T>::Set()` (described here) or `ON_CALL()`.

Please note that once you expressed interest in a particular mock method (via `EXPECT_CALL()`), all invocations to it must match some expectation. If this function is called but the arguments don't match any `EXPECT_CALL()` statement, it will be an error.

Disallowing Unexpected Calls

If a mock method shouldn't be called at all, explicitly say so:

```
using ::testing::_;
...
  EXPECT_CALL(foo, Bar(_))
      .Times(0);
```

If some calls to the method are allowed, but the rest are not, just list all the expected calls:

```
using ::testing::AnyNumber;
using ::testing::Gt;
...
  EXPECT_CALL(foo, Bar(5));
  EXPECT_CALL(foo, Bar(Gt(10)))
      .Times(AnyNumber());
```

A call to `foo.Bar()` that doesn't match any of the `EXPECT_CALL()` statements will be an error.

Understanding Uninteresting vs Unexpected Calls {#uninteresting-vs-unexpected}

Uninteresting calls and *unexpected* calls are different concepts in gMock. *Very* different.

A call `x.Y(...)` is **uninteresting** if there's *not even a single* `EXPECT_CALL(x, Y(...))` set. In other words, the test isn't interested in the `x.Y()` method at all, as evident in that the test doesn't care to say anything about it.

A call `x.Y(...)` is **unexpected** if there are *some* `EXPECT_CALL(x, Y(...))`s set, but none of them matches the call. Put another way, the test is interested in the `x.Y()` method (therefore it explicitly sets some `EXPECT_CALL` to verify how it's called); however, the verification fails as the test doesn't expect this particular call to happen.

An unexpected call is always an error, as the code under test doesn't behave the way the test expects it to behave.

By default, an uninteresting call is not an error, as it violates no constraint specified by the test. (gMock's philosophy is that saying nothing means there is no constraint.) However, it leads to a warning, as it *might* indicate a problem (e.g. the test author might have forgotten to specify a constraint).

In gMock, `NiceMock` and `StrictMock` can be used to make a mock class "nice" or "strict". How does this affect uninteresting calls and unexpected calls?

A **nice mock** suppresses uninteresting call *warnings*. It is less chatty than the default mock, but otherwise is the same. If a test fails with a default mock, it will also fail using a nice mock instead. And vice versa. Don't expect making a mock nice to change the test's result.

A **strict mock** turns uninteresting call warnings into errors. So making a mock strict may change the test's result.

Let's look at an example:

```
TEST(...) {
  NiceMock<MockDomainRegistry> mock_registry;
  EXPECT_CALL(mock_registry, GetDomainOwner("google.com"))
          .WillRepeatedly(Return("Larry Page"));

  // Use mock_registry in code under test.
  ... &mock_registry ...
}
```

The sole `EXPECT_CALL` here says that all calls to `GetDomainOwner()` must have `"google.com"` as the argument. If `GetDomainOwner("yahoo.com")` is called, it will be an unexpected call, and thus an error. *Having a nice mock doesn't change the severity of an unexpected call.*

So how do we tell gMock that `GetDomainOwner()` can be called with some other arguments as well? The standard technique is to add a "catch all" `EXPECT_CALL`:

```
  EXPECT_CALL(mock_registry, GetDomainOwner(_))
        .Times(AnyNumber());  // catches all other calls to this method.
  EXPECT_CALL(mock_registry, GetDomainOwner("google.com"))
        .WillRepeatedly(Return("Larry Page"));
```

Remember that `_` is the wildcard matcher that matches anything. With this, if `GetDomainOwner("google.com")` is called, it will do what the second `EXPECT_CALL` says; if it is called with a different argument, it will do what the first `EXPECT_CALL` says.

Note that the order of the two `EXPECT_CALL`s is important, as a newer `EXPECT_CALL` takes precedence over an older one.

For more on uninteresting calls, nice mocks, and strict mocks, read ["The Nice, the Strict, and the Naggy"].

Ignoring Uninteresting Arguments {#ParameterlessExpectations}

If your test doesn't care about the parameters (it only cares about the number or order of calls), you can often simply omit the parameter list:

```
// Expect foo.Bar( ... ) twice with any arguments.
EXPECT_CALL(foo, Bar).Times(2);

// Delegate to the given method whenever the factory is invoked.
ON_CALL(foo_factory, MakeFoo)
    .WillByDefault(&BuildFooForTest);
```

This functionality is only available when a method is not overloaded; to prevent unexpected behavior it is a compilation error to try to set an expectation on a method where the specific overload is ambiguous. You can work around this by supplying a [simpler mock interface] than the mocked class provides.

This pattern is also useful when the arguments are interesting, but match logic is substantially complex. You can leave the argument list unspecified and use SaveArg actions to [save the values for later verification]. If you do that, you can easily differentiate calling the method the wrong number of times from calling it with the wrong arguments.

Expecting Ordered Calls {#OrderedCalls}

Although an `EXPECT_CALL()` statement defined later takes precedence when gMock tries to match a function call with an expectation, by default calls don't have to happen in the order `EXPECT_CALL()` statements are written. For example, if the arguments match the matchers in the second `EXPECT_CALL()`, but not those in the first and third, then the second expectation will be used.

If you would rather have all calls occur in the order of the expectations, put the `EXPECT_CALL()` statements in a block where you define a variable of type `InSequence`:

```
using ::testing::_;
using ::testing::InSequence;

  {
    InSequence s;

    EXPECT_CALL(foo, DoThis(5));
    EXPECT_CALL(bar, DoThat(_))
        .Times(2);
    EXPECT_CALL(foo, DoThis(6));
  }
```

In this example, we expect a call to `foo.DoThis(5)`, followed by two calls to `bar.DoThat()` where the argument can be anything, which are in turn followed by a call to `foo.DoThis(6)`. If a call occurred out-of-order, gMock will report an error.

Expecting Partially Ordered Calls {#PartialOrder}

Sometimes requiring everything to occur in a predetermined order can lead to brittle tests. For example, we may care about `A` occurring before both `B` and `C`, but aren't interested in the relative order of `B` and `C`. In this case, the test should reflect our real intent, instead of being overly constraining.

gMock allows you to impose an arbitrary DAG (directed acyclic graph) on the calls. One way to express the DAG is to use the [`After` clause] of `EXPECT_CALL`.

Another way is via the `InSequence()` clause (not the same as the `InSequence` class), which we borrowed from jMock 2. It's less flexible than `After()`, but more convenient when you have long chains of sequential calls, as it doesn't require you to come up with different names for the expectations in the chains. Here's how it works:

If we view `EXPECT_CALL()` statements as nodes in a graph, and add an edge from node A to node B wherever A must occur before B, we can get a DAG. We use the term "sequence" to mean a directed path in this DAG. Now, if we decompose the DAG into sequences, we just need to know which sequences each `EXPECT_CALL()` belongs to in order to be able to reconstruct the original DAG.

So, to specify the partial order on the expectations we need to do two things: first to define some `Sequence` objects, and then for each `EXPECT_CALL()` say which `Sequence` objects it is part of.

Expectations in the same sequence must occur in the order they are written. For example,

```
using ::testing::Sequence;
...
  Sequence s1, s2;

  EXPECT_CALL(foo, A())
      .InSequence(s1, s2);
  EXPECT_CALL(bar, B())
      .InSequence(s1);
  EXPECT_CALL(bar, C())
      .InSequence(s2);
  EXPECT_CALL(foo, D())
      .InSequence(s2);
```

specifies the following DAG (where `s1` is `A -> B`, and `s2` is `A -> C -> D`):

```
         +---> B
         |
  A ---|
         |
         +---> C ---> D
```

This means that A must occur before B and C, and C must occur before D. There's no restriction about the order other than these.

Controlling When an Expectation Retires

When a mock method is called, gMock only considers expectations that are still active. An expectation is active when created, and becomes inactive (aka *retires*) when a call that has to occur later has occurred. For example, in

```
using ::testing::_;
using ::testing::Sequence;
...
  Sequence s1, s2;

  EXPECT_CALL(log, Log(WARNING, _, "File too large."))      // #1
      .Times(AnyNumber())
      .InSequence(s1, s2);
  EXPECT_CALL(log, Log(WARNING, _, "Data set is empty."))   // #2
      .InSequence(s1);
  EXPECT_CALL(log, Log(WARNING, _, "User not found."))      // #3
      .InSequence(s2);
```

as soon as either #2 or #3 is matched, #1 will retire. If a warning `"File too large."` is logged after this, it will be an error.

Note that an expectation doesn't retire automatically when it's saturated. For example,

```
using ::testing::_;
...
  EXPECT_CALL(log, Log(WARNING, _, _));                     // #1
  EXPECT_CALL(log, Log(WARNING, _, "File too large."));     // #2
```

says that there will be exactly one warning with the message `"File too large."`. If the second warning contains this message too, #2 will match again and result in an upper-bound-violated error.

If this is not what you want, you can ask an expectation to retire as soon as it becomes saturated:

```
using ::testing::_;
...
  EXPECT_CALL(log, Log(WARNING, _, _));                     // #1
  EXPECT_CALL(log, Log(WARNING, _, "File too large."))      // #2
      .RetiresOnSaturation();
```

Here #2 can be used only once, so if you have two warnings with the message `"File too large."`, the first will match #2 and the second will match #1 - there will be no error.

Using Actions

Returning References from Mock Methods

If a mock function's return type is a reference, you need to use `ReturnRef()` instead of `Return()` to return a result:

```
using ::testing::ReturnRef;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(Bar&, GetBar, (), (override));
};
...
  MockFoo foo;
  Bar bar;
  EXPECT_CALL(foo, GetBar())
      .WillOnce(ReturnRef(bar));
...
```

Returning Live Values from Mock Methods

The `Return(x)` action saves a copy of `x` when the action is created, and always returns the same value whenever it's executed. Sometimes you may want to instead return the *live* value of `x` (i.e. its value at the time when the action is *executed*.). Use either `ReturnRef()` or `ReturnPointee()` for this purpose.

If the mock function's return type is a reference, you can do it using `ReturnRef(x)`, as shown in the previous recipe ("Returning References from Mock Methods"). However, gMock doesn't let you use `ReturnRef()` in a mock function whose return type is not a reference, as doing that usually indicates a user error. So, what shall you do?

Though you may be tempted, DO NOT use `std::ref()`:

```
using testing::Return;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, GetValue, (), (override));
};
...
  int x = 0;
  MockFoo foo;
  EXPECT_CALL(foo, GetValue())
      .WillRepeatedly(Return(std::ref(x)));  // Wrong!
  x = 42;
  EXPECT_EQ(42, foo.GetValue());
```

Unfortunately, it doesn't work here. The above code will fail with error:

```
Value of: foo.GetValue()
  Actual: 0
Expected: 42
```

The reason is that `Return(*value*)` converts `value` to the actual return type of the mock function at the time when the action is *created*, not when it is *executed*. (This behavior was chosen for the action to be safe when `value` is a proxy object that references some temporary objects.) As a result, `std::ref(x)` is converted to an `int` value (instead of a `const int&`) when the expectation is set, and `Return(std::ref(x))` will always return 0.

`ReturnPointee(pointer)` was provided to solve this problem specifically. It returns the value pointed to by `pointer` at the time the action is *executed*:

```
using testing::ReturnPointee;
...
  int x = 0;
  MockFoo foo;
  EXPECT_CALL(foo, GetValue())
      .WillRepeatedly(ReturnPointee(&x));  // Note the & here.
  x = 42;
  EXPECT_EQ(42, foo.GetValue());  // This will succeed now.
```

Combining Actions

Want to do more than one thing when a function is called? That's fine. `DoAll()` allow you to do sequence of actions every time. Only the return value of the last action in the sequence will be used.

```
using ::testing::_;
using ::testing::DoAll;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(bool, Bar, (int n), (override));
};
...
  EXPECT_CALL(foo, Bar(_))
      .WillOnce(DoAll(action_1,
                      action_2,
                      ...
                      action_n));
```

Verifying Complex Arguments {#SaveArgVerify}

If you want to verify that a method is called with a particular argument but the match criteria is complex, it can be difficult to distinguish between cardinality failures (calling the method the wrong number of times) and argument match failures. Similarly, if you are matching multiple parameters, it may not be easy to distinguishing which argument failed to match. For example:

```
  // Not ideal: this could fail because of a problem with arg1 or arg2, or maybe
  // just the method wasn't called.
  EXPECT_CALL(foo, SendValues(_, ElementsAre(1, 4, 4, 7), EqualsProto( ... )));
```

You can instead save the arguments and test them individually:

```
  EXPECT_CALL(foo, SendValues)
      .WillOnce(DoAll(SaveArg<1>(&actual_array), SaveArg<2>(&actual_proto)));
  ... run the test
  EXPECT_THAT(actual_array, ElementsAre(1, 4, 4, 7));
  EXPECT_THAT(actual_proto, EqualsProto( ... ));
```

Mocking Side Effects {#MockingSideEffects}

Sometimes a method exhibits its effect not via returning a value but via side effects. For example, it may change some global state or modify an output argument. To mock side effects, in general you can define your own action by implementing `::testing::ActionInterface`.

If all you need to do is to change an output argument, the built-in `SetArgPointee()` action is convenient:

```
using ::testing::_;
using ::testing::SetArgPointee;

class MockMutator : public Mutator {
 public:
  MOCK_METHOD(void, Mutate, (bool mutate, int* value), (override));
  ...
}
...
  MockMutator mutator;
  EXPECT_CALL(mutator, Mutate(true, _))
      .WillOnce(SetArgPointee<1>(5));
```

In this example, when `mutator.Mutate()` is called, we will assign 5 to the `int` variable pointed to by argument #1 (0-based).

`SetArgPointee()` conveniently makes an internal copy of the value you pass to it, removing the need to keep the value in scope and alive. The implication however is that the value must have a copy constructor and assignment operator.

If the mock method also needs to return a value as well, you can chain `SetArgPointee()` with `Return()` using `DoAll()`, remembering to put the `Return()` statement last:

```
using ::testing::_;
using ::testing::DoAll;
using ::testing::Return;
using ::testing::SetArgPointee;

class MockMutator : public Mutator {
 public:
  ...
  MOCK_METHOD(bool, MutateInt, (int* value), (override));
}
...
  MockMutator mutator;
  EXPECT_CALL(mutator, MutateInt(_))
      .WillOnce(DoAll(SetArgPointee<0>(5),
                      Return(true)));
```

Note, however, that if you use the `ReturnOKWith()` method, it will override the values provided by `SetArgPointee()` in the response parameters of your function call.

If the output argument is an array, use the `SetArrayArgument<N>(first, last)` action instead. It copies the elements in source range `[first, last)` to the array pointed to by the `N`-th (0-based) argument:

```
using ::testing::NotNull;
using ::testing::SetArrayArgument;

class MockArrayMutator : public ArrayMutator {
 public:
  MOCK_METHOD(void, Mutate, (int* values, int num_values), (override));
  ...
}
...
  MockArrayMutator mutator;
  int values[5] = {1, 2, 3, 4, 5};
  EXPECT_CALL(mutator, Mutate(NotNull(), 5))
      .WillOnce(SetArrayArgument<0>(values, values + 5));
```

This also works when the argument is an output iterator:

```
using ::testing::_;
using ::testing::SetArrayArgument;

class MockRolodex : public Rolodex {
 public:
  MOCK_METHOD(void, GetNames, (std::back_insert_iterator<vector<string>>),
              (override));
  ...
}
...
  MockRolodex rolodex;
  vector<string> names = {"George", "John", "Thomas"};
  EXPECT_CALL(rolodex, GetNames(_))
      .WillOnce(SetArrayArgument<0>(names.begin(), names.end()));
```

Changing a Mock Object's Behavior Based on the State

If you expect a call to change the behavior of a mock object, you can use `::testing::InSequence` to specify different behaviors before and after the call:

```
using ::testing::InSequence;
using ::testing::Return;

...
  {
     InSequence seq;
     EXPECT_CALL(my_mock, IsDirty())
         .WillRepeatedly(Return(true));
     EXPECT_CALL(my_mock, Flush());
     EXPECT_CALL(my_mock, IsDirty())
         .WillRepeatedly(Return(false));
  }
  my_mock.FlushIfDirty();
```

This makes `my_mock.IsDirty()` return `true` before `my_mock.Flush()` is called and return `false` afterwards.

If the behavior change is more complex, you can store the effects in a variable and make a mock method get its return value from that variable:

```
using ::testing::_;
using ::testing::SaveArg;
using ::testing::Return;

ACTION_P(ReturnPointee, p) { return *p; }
...
  int previous_value = 0;
  EXPECT_CALL(my_mock, GetPrevValue)
      .WillRepeatedly(ReturnPointee(&previous_value));
  EXPECT_CALL(my_mock, UpdateValue)
      .WillRepeatedly(SaveArg<0>(&previous_value));
  my_mock.DoSomethingToUpdateValue();
```

Here `my_mock.GetPrevValue()` will always return the argument of the last `UpdateValue()` call.

Setting the Default Value for a Return Type {#DefaultValue}

If a mock method's return type is a built-in C++ type or pointer, by default it will return 0 when invoked. Also, in C++ 11 and above, a mock method whose return type has a default constructor will return a default-constructed value by default. You only need to specify an action if this default value doesn't work for you.

Sometimes, you may want to change this default value, or you may want to specify a default value for types gMock doesn't know about. You can do this using the `::testing::DefaultValue` class template:

```
using ::testing::DefaultValue;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(Bar, CalculateBar, (), (override));
};



...
  Bar default_bar;
  // Sets the default return value for type Bar.
  DefaultValue<Bar>::Set(default_bar);

  MockFoo foo;

  // We don't need to specify an action here, as the default
  // return value works for us.
  EXPECT_CALL(foo, CalculateBar());

  foo.CalculateBar();  // This should return default_bar.

  // Unsets the default return value.
  DefaultValue<Bar>::Clear();
```

Please note that changing the default value for a type can make your tests hard to understand. We recommend you to use this feature judiciously. For example, you may want to make sure the `Set()` and `Clear()` calls are right next to the code that uses your mock.

Setting the Default Actions for a Mock Method

You've learned how to change the default value of a given type. However, this may be too coarse for your purpose: perhaps you have two mock methods with the same return type and you want them to have different behaviors. The `ON_CALL()` macro allows you to customize your mock's behavior at the method level:

```
using ::testing::_;
using ::testing::AnyNumber;
using ::testing::Gt;
using ::testing::Return;
...
  ON_CALL(foo, Sign(_))
      .WillByDefault(Return(-1));
  ON_CALL(foo, Sign(0))
      .WillByDefault(Return(0));
  ON_CALL(foo, Sign(Gt(0)))
      .WillByDefault(Return(1));

  EXPECT_CALL(foo, Sign(_))
      .Times(AnyNumber());

  foo.Sign(5);   // This should return 1.
  foo.Sign(-9);  // This should return -1.
  foo.Sign(0);   // This should return 0.
```

As you may have guessed, when there are more than one `ON_CALL()` statements, the newer ones in the order take precedence over the older ones. In other words, the **last** one that matches the function arguments will be used. This matching order allows you to set up the common behavior in a mock object's constructor or the test fixture's set-up phase and specialize the mock's behavior later.

Note that both `ON_CALL` and `EXPECT_CALL` have the same "later statements take precedence" rule, but they don't interact. That is, `EXPECT_CALL`s have their own precedence order distinct from the `ON_CALL` precedence order.

Using Functions/Methods/Functors/Lambdas as Actions {#FunctionsAsActions}

If the built-in actions don't suit you, you can use an existing callable (function, `std::function`, method, functor, lambda) as an action.

```
using ::testing::_; using ::testing::Invoke;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, Sum, (int x, int y), (override));
  MOCK_METHOD(bool, ComplexJob, (int x), (override));
};

int CalculateSum(int x, int y) { return x + y; }
int Sum3(int x, int y, int z) { return x + y + z; }

class Helper {
 public:
  bool ComplexJob(int x);
};

...
  MockFoo foo;
  Helper helper;
  EXPECT_CALL(foo, Sum(_, _))
      .WillOnce(&CalculateSum)
      .WillRepeatedly(Invoke(NewPermanentCallback(Sum3, 1)));
  EXPECT_CALL(foo, ComplexJob(_))
      .WillOnce(Invoke(&helper, &Helper::ComplexJob))
      .WillOnce([] { return true; })
      .WillRepeatedly([](int x) { return x > 0; });

  foo.Sum(5, 6);         // Invokes CalculateSum(5, 6).
  foo.Sum(2, 3);         // Invokes Sum3(1, 2, 3).
  foo.ComplexJob(10);    // Invokes helper.ComplexJob(10).
  foo.ComplexJob(-1);    // Invokes the inline lambda.
```

The only requirement is that the type of the function, etc must be *compatible* with the signature of the mock function, meaning that the latter's arguments (if it takes any) can be implicitly converted to the corresponding arguments of the former, and the former's return type can be implicitly converted to that of the latter. So, you can invoke something whose type is *not* exactly the same as the mock function, as long as it's safe to do so - nice, huh?

Note that:

- The action takes ownership of the callback and will delete it when the action itself is destructed.
- If the type of a callback is derived from a base callback type `C`, you need to implicitly cast it to `C` to resolve the overloading, e.g.

  ```
  using ::testing::Invoke;
  ...
    ResultCallback<bool>* is_ok = ...;
    ... Invoke(is_ok) ...;  // This works.

    BlockingClosure* done = new BlockingClosure;
    ... Invoke(implicit_cast<Closure*>(done)) ...;  // The cast is necessary.
  ```

Using Functions with Extra Info as Actions

The function or functor you call using `Invoke()` must have the same number of arguments as the mock function you use it for. Sometimes you may have a function that takes more arguments, and you are willing to pass in the extra arguments yourself to fill the gap. You can do this in gMock using callbacks with pre-bound arguments. Here's an example:

```
using ::testing::Invoke;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(char, DoThis, (int n), (override));
};

char SignOfSum(int x, int y) {
  const int sum = x + y;
  return (sum > 0) ? '+' : (sum < 0) ? '-' : '0';
}

TEST_F(FooTest, Test) {
  MockFoo foo;

  EXPECT_CALL(foo, DoThis(2))
      .WillOnce(Invoke(NewPermanentCallback(SignOfSum, 5)));
  EXPECT_EQ('+', foo.DoThis(2));  // Invokes SignOfSum(5, 2).
}
```

Invoking a Function/Method/Functor/Lambda/Callback Without Arguments

`Invoke()` passes the mock function's arguments to the function, etc being invoked such that the callee has the full context of the call to work with. If the invoked function is not interested in some or all of the arguments, it can simply ignore them.

Yet, a common pattern is that a test author wants to invoke a function without the arguments of the mock function. She could do that using a wrapper function that throws away the arguments before invoking an underlining nullary function. Needless to say, this can be tedious and obscures the intent of the test.

There are two solutions to this problem. First, you can pass any callable of zero args as an action. Alternatively, use `InvokeWithoutArgs()`, which is like `Invoke()` except that it doesn't pass the mock function's arguments to the callee. Here's an example of each:

```
using ::testing::_;
using ::testing::InvokeWithoutArgs;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(bool, ComplexJob, (int n), (override));
};
```

```
bool Job1() { ... }
bool Job2(int n, char c) { ... }

...
  MockFoo foo;
  EXPECT_CALL(foo, ComplexJob(_))
      .WillOnce([] { Job1(); });
      .WillOnce(InvokeWithoutArgs(NewPermanentCallback(Job2, 5, 'a')));

  foo.ComplexJob(10);  // Invokes Job1().
  foo.ComplexJob(20);  // Invokes Job2(5, 'a').
```

Note that:

- The action takes ownership of the callback and will delete it when the action itself is destructed.
- If the type of a callback is derived from a base callback type `C`, you need to implicitly cast it to `C` to resolve the overloading, e.g.

  ```
  using ::testing::InvokeWithoutArgs;
  ...
    ResultCallback<bool>* is_ok = ...;
    ... InvokeWithoutArgs(is_ok) ...;  // This works.

    BlockingClosure* done = ...;
    ... InvokeWithoutArgs(implicit_cast<Closure*>(done)) ...;
    // The cast is necessary.
  ```

Invoking an Argument of the Mock Function

Sometimes a mock function will receive a function pointer, a functor (in other words, a "callable") as an argument, e.g.

```
class MockFoo : public Foo {
 public:
  MOCK_METHOD(bool, DoThis, (int n, (ResultCallback1<bool, int>* callback)),
              (override));
};
```

and you may want to invoke this callable argument:

```
using ::testing::_;
...
  MockFoo foo;
  EXPECT_CALL(foo, DoThis(_, _))
      .WillOnce(...);
      // Will execute callback->Run(5), where callback is the
      // second argument DoThis() receives.
```

{: .callout .note}
NOTE: The section below is legacy documentation from before C++ had lambdas:

Arghh, you need to refer to a mock function argument but C++ has no lambda (yet), so you have to define your own action. :-(Or do you really?

Well, gMock has an action to solve *exactly* this problem:

```
InvokeArgument<N>(arg_1, arg_2, ..., arg_m)
```

will invoke the `N`-th (0-based) argument the mock function receives, with `arg_1`, `arg_2`, ..., and `arg_m`. No matter if the argument is a function pointer, a functor, or a callback. gMock handles them all.

With that, you could write:

```
using ::testing::_;
using ::testing::InvokeArgument;
...
  EXPECT_CALL(foo, DoThis(_, _))
      .WillOnce(InvokeArgument<1>(5));
      // Will execute callback->Run(5), where callback is the
      // second argument DoThis() receives.
```

What if the callable takes an argument by reference? No problem - just wrap it inside `std::ref()`:

```
  ...
  MOCK_METHOD(bool, Bar,
              ((ResultCallback2<bool, int, const Helper&>* callback)),
              (override));
  ...
  using ::testing::_;
  using ::testing::InvokeArgument;
  ...
  MockFoo foo;
  Helper helper;
  ...
  EXPECT_CALL(foo, Bar(_))
      .WillOnce(InvokeArgument<0>(5, std::ref(helper)));
      // std::ref(helper) guarantees that a reference to helper, not a copy of
      // it, will be passed to the callback.
```

What if the callable takes an argument by reference and we do **not** wrap the argument in `std::ref()`? Then `InvokeArgument()` will *make a copy* of the argument, and pass a *reference to the copy*, instead of a reference to the original value, to the callable. This is especially handy when the argument is a temporary value:

```
  ...
  MOCK_METHOD(bool, DoThat, (bool (*f)(const double& x, const string& s)),
              (override));
  ...
  using ::testing::_;
  using ::testing::InvokeArgument;
  ...
  MockFoo foo;
  ...
  EXPECT_CALL(foo, DoThat(_))
      .WillOnce(InvokeArgument<0>(5.0, string("Hi")));
      // Will execute (*f)(5.0, string("Hi")), where f is the function pointer
      // DoThat() receives.  Note that the values 5.0 and string("Hi") are
      // temporary and dead once the EXPECT_CALL() statement finishes.  Yet
      // it's fine to perform this action later, since a copy of the values
      // are kept inside the InvokeArgument action.
```

Ignoring an Action's Result

Sometimes you have an action that returns *something*, but you need an action that returns `void` (perhaps you want to use it in a mock function that returns `void`, or perhaps it needs to be used in `DoAll()` and it's not the last in the list). `IgnoreResult()` lets you do that. For example:

```
using ::testing::_;
using ::testing::DoAll;
using ::testing::IgnoreResult;
using ::testing::Return;

int Process(const MyData& data);
string DoSomething();

class MockFoo : public Foo {
 public:
  MOCK_METHOD(void, Abc, (const MyData& data), (override));
  MOCK_METHOD(bool, Xyz, (), (override));
};

  ...
  MockFoo foo;
  EXPECT_CALL(foo, Abc(_))
      // .WillOnce(Invoke(Process));
      // The above line won't compile as Process() returns int but Abc() needs
      // to return void.
      .WillOnce(IgnoreResult(Process));
  EXPECT_CALL(foo, Xyz())
      .WillOnce(DoAll(IgnoreResult(DoSomething),
                      // Ignores the string DoSomething() returns.
                      Return(true)));
```

Note that you **cannot** use `IgnoreResult()` on an action that already returns `void`. Doing so will lead to ugly compiler errors.

Selecting an Action's Arguments {#SelectingArgs}

Say you have a mock function `Foo()` that takes seven arguments, and you have a custom action that you want to invoke when `Foo()` is called. Trouble is, the custom action only wants three arguments:

```
using ::testing::_;
using ::testing::Invoke;
...
  MOCK_METHOD(bool, Foo,
              (bool visible, const string& name, int x, int y,
               (const map<pair<int, int>>), double& weight, double min_weight,
               double max_wight));
...
bool IsVisibleInQuadrant1(bool visible, int x, int y) {
  return visible && x >= 0 && y >= 0;
}
...
  EXPECT_CALL(mock, Foo)
      .WillOnce(Invoke(IsVisibleInQuadrant1));  // Uh, won't compile. :-(
```

To please the compiler God, you need to define an "adaptor" that has the same signature as `Foo()` and calls the custom action with the right arguments:

```
using ::testing::_;
using ::testing::Invoke;
...
bool MyIsVisibleInQuadrant1(bool visible, const string& name, int x, int y,
                            const map<pair<int, int>, double>& weight,
                            double min_weight, double max_wight) {
  return IsVisibleInQuadrant1(visible, x, y);
}
...
  EXPECT_CALL(mock, Foo)
      .WillOnce(Invoke(MyIsVisibleInQuadrant1));  // Now it works.
```

But isn't this awkward?

gMock provides a generic *action adaptor*, so you can spend your time minding more important business than writing your own adaptors. Here's the syntax:

```
WithArgs<N1, N2, ..., Nk>(action)
```

creates an action that passes the arguments of the mock function at the given indices (0-based) to the inner `action` and performs it. Using `WithArgs`, our original example can be written as:

```
using ::testing::_;
using ::testing::Invoke;
using ::testing::WithArgs;
...
  EXPECT_CALL(mock, Foo)
      .WillOnce(WithArgs<0, 2, 3>(Invoke(IsVisibleInQuadrant1)));  // No need to
define your own adaptor.
```

For better readability, gMock also gives you:

- `WithoutArgs(action)` when the inner `action` takes *no* argument, and
- `WithArg<N>(action)` (no `s` after `Arg`) when the inner `action` takes *one* argument.

As you may have realized, `InvokeWithoutArgs(...)` is just syntactic sugar for `WithoutArgs(Invoke(...))`.

Here are more tips:

- The inner action used in `WithArgs` and friends does not have to be `Invoke()` -- it can be anything.
- You can repeat an argument in the argument list if necessary, e.g. `WithArgs<2, 3, 3, 5>(...)`.
- You can change the order of the arguments, e.g. `WithArgs<3, 2, 1>(...)`.
- The types of the selected arguments do *not* have to match the signature of the inner action exactly. It works as long as they can be implicitly converted to the corresponding arguments of the inner action. For example, if the 4-th argument of the mock function is an `int` and `my_action` takes a `double`, `WithArg<4>(my_action)` will work.

Ignoring Arguments in Action Functions

The selecting-an-action's-arguments recipe showed us one way to make a mock function and an action with incompatible argument lists fit together. The downside is that wrapping the action in `WithArgs<...>()` can get tedious for people writing the tests.

If you are defining a function (or method, functor, lambda, callback) to be used with `Invoke*()`, and you are not interested in some of its arguments, an alternative to `WithArgs` is to declare the uninteresting arguments as `Unused`. This makes the definition less cluttered and less fragile in case the types of the uninteresting arguments change. It could also increase the chance the action function can be reused. For example, given

```
 public:
  MOCK_METHOD(double, Foo, double(const string& label, double x, double y),
              (override));
  MOCK_METHOD(double, Bar, (int index, double x, double y), (override));
```

instead of

```
using ::testing::_;
using ::testing::Invoke;

double DistanceToOriginWithLabel(const string& label, double x, double y) {
  return sqrt(x*x + y*y);
}
double DistanceToOriginWithIndex(int index, double x, double y) {
  return sqrt(x*x + y*y);
}
...
  EXPECT_CALL(mock, Foo("abc", _, _))
      .WillOnce(Invoke(DistanceToOriginWithLabel));
  EXPECT_CALL(mock, Bar(5, _, _))
      .WillOnce(Invoke(DistanceToOriginWithIndex));
```

you could write

```
using ::testing::_;
using ::testing::Invoke;
using ::testing::Unused;

double DistanceToOrigin(Unused, double x, double y) {
  return sqrt(x*x + y*y);
}
...
  EXPECT_CALL(mock, Foo("abc", _, _))
      .WillOnce(Invoke(DistanceToOrigin));
  EXPECT_CALL(mock, Bar(5, _, _))
      .WillOnce(Invoke(DistanceToOrigin));
```

Sharing Actions

Just like matchers, a gMock action object consists of a pointer to a ref-counted implementation object. Therefore copying actions is also allowed and very efficient. When the last action that references the implementation object dies, the implementation object will be deleted.

If you have some complex action that you want to use again and again, you may not have to build it from scratch every time. If the action doesn't have an internal state (i.e. if it always does the same thing no matter how many times it has been called), you can assign it to an action variable and use that variable repeatedly. For example:

```
using ::testing::Action;
using ::testing::DoAll;
using ::testing::Return;
using ::testing::SetArgPointee;
...
  Action<bool(int*)> set_flag = DoAll(SetArgPointee<0>(5),
                                      Return(true));
  ... use set_flag in .WillOnce() and .WillRepeatedly() ...
```

However, if the action has its own state, you may be surprised if you share the action object. Suppose you have an action factory `IncrementCounter(init)` which creates an action that increments and returns a counter whose initial value is `init`, using two actions created from the same expression and using a shared action will exhibit different behaviors. Example:

```
  EXPECT_CALL(foo, DoThis())
      .WillRepeatedly(IncrementCounter(0));
  EXPECT_CALL(foo, DoThat())
      .WillRepeatedly(IncrementCounter(0));
  foo.DoThis();  // Returns 1.
  foo.DoThis();  // Returns 2.
  foo.DoThat();  // Returns 1 - Blah() uses a different
                 // counter than Bar()'s.
```

versus

```
using ::testing::Action;
...
  Action<int()> increment = IncrementCounter(0);
  EXPECT_CALL(foo, DoThis())
      .WillRepeatedly(increment);
  EXPECT_CALL(foo, DoThat())
      .WillRepeatedly(increment);
  foo.DoThis();  // Returns 1.
  foo.DoThis();  // Returns 2.
  foo.DoThat();  // Returns 3 - the counter is shared.
```

Testing Asynchronous Behavior

One oft-encountered problem with gMock is that it can be hard to test asynchronous behavior. Suppose you had a `EventQueue` class that you wanted to test, and you created a separate `EventDispatcher` interface so that you could easily mock it out. However, the implementation of the class fired all the events on a background thread, which made test timings difficult. You could just insert `sleep()` statements and hope for the best, but that makes your test behavior nondeterministic. A better way is to use gMock actions and `Notification` objects to force your asynchronous test to behave synchronously.

```
class MockEventDispatcher : public EventDispatcher {
  MOCK_METHOD(bool, DispatchEvent, (int32), (override));
};

TEST(EventQueueTest, EnqueueEventTest) {
  MockEventDispatcher mock_event_dispatcher;
  EventQueue event_queue(&mock_event_dispatcher);

  const int32 kEventId = 321;
  absl::Notification done;
  EXPECT_CALL(mock_event_dispatcher, DispatchEvent(kEventId))
      .WillOnce([&done] { done.Notify(); });

  event_queue.EnqueueEvent(kEventId);
  done.WaitForNotification();
}
```

In the example above, we set our normal gMock expectations, but then add an additional action to notify the `Notification` object. Now we can just call `Notification::WaitForNotification()` in the main thread to wait for the asynchronous call to finish. After that, our test suite is complete and we can safely exit.

{: .callout .note}
Note: this example has a downside: namely, if the expectation is not satisfied, our test will run forever. It will eventually time-out and fail, but it will take longer and be slightly harder to debug. To alleviate this problem, you can use `WaitForNotificationWithTimeout(ms)` instead of `WaitForNotification()`.

Misc Recipes on Using gMock

Mocking Methods That Use Move-Only Types

C++11 introduced *move-only types*. A move-only-typed value can be moved from one object to another, but cannot be copied. `std::unique_ptr<T>` is probably the most commonly used move-only type.

Mocking a method that takes and/or returns move-only types presents some challenges, but nothing insurmountable. This recipe shows you how you can do it. Note that the support for move-only method arguments was only introduced to gMock in April 2017; in older code, you may find more complex workarounds for lack of this feature.

Let's say we are working on a fictional project that lets one post and share snippets called "buzzes". Your code uses these types:

```
enum class AccessLevel { kInternal, kPublic };

class Buzz {
 public:
  explicit Buzz(AccessLevel access) { ... }
  ...
};

class Buzzer {
 public:
  virtual ~Buzzer() {}
  virtual std::unique_ptr<Buzz> MakeBuzz(StringPiece text) = 0;
  virtual bool ShareBuzz(std::unique_ptr<Buzz> buzz, int64_t timestamp) = 0;
  ...
};
```

A `Buzz` object represents a snippet being posted. A class that implements the `Buzzer` interface is capable of creating and sharing `Buzz`es. Methods in `Buzzer` may return a `unique_ptr<Buzz>` or take a `unique_ptr<Buzz>`. Now we need to mock `Buzzer` in our tests.

To mock a method that accepts or returns move-only types, you just use the familiar `MOCK_METHOD` syntax as usual:

```
class MockBuzzer : public Buzzer {
 public:
  MOCK_METHOD(std::unique_ptr<Buzz>, MakeBuzz, (StringPiece text), (override));
  MOCK_METHOD(bool, ShareBuzz, (std::unique_ptr<Buzz> buzz, int64_t timestamp),
              (override));
};
```

Now that we have the mock class defined, we can use it in tests. In the following code examples, we assume that we have defined a `MockBuzzer` object named `mock_buzzer_`:

```
  MockBuzzer mock_buzzer_;
```

First let's see how we can set expectations on the `MakeBuzz()` method, which returns a `unique_ptr<Buzz>`.

As usual, if you set an expectation without an action (i.e. the `.WillOnce()` or `.WillRepeatedly()` clause), when that expectation fires, the default action for that method will be taken. Since `unique_ptr<>` has a default constructor that returns a null `unique_ptr`, that's what you'll get if you don't specify an action:

```
  // Use the default action.
  EXPECT_CALL(mock_buzzer_, MakeBuzz("hello"));

  // Triggers the previous EXPECT_CALL.
  EXPECT_EQ(nullptr, mock_buzzer_.MakeBuzz("hello"));
```

If you are not happy with the default action, you can tweak it as usual; see Setting Default Actions.

If you just need to return a pre-defined move-only value, you can use the `Return(ByMove(...))` action:

```
  // When this fires, the unique_ptr<> specified by ByMove(...) will
  // be returned.
  EXPECT_CALL(mock_buzzer_, MakeBuzz("world"))
      .WillOnce(Return(ByMove(MakeUnique<Buzz>(AccessLevel::kInternal))));

  EXPECT_NE(nullptr, mock_buzzer_.MakeBuzz("world"));
```

Note that `ByMove()` is essential here - if you drop it, the code won't compile.

Quiz time! What do you think will happen if a `Return(ByMove(...))` action is performed more than once (e.g. you write `... .WillRepeatedly(Return(ByMove(...)));`)? Come think of it, after the first time the action runs, the source value will be consumed (since it's a move-only value), so the next time around, there's no value to move from -- you'll get a run-time error that `Return(ByMove(...))` can only be run once.

If you need your mock method to do more than just moving a pre-defined value, remember that you can always use a lambda or a callable object, which can do pretty much anything you want:

```
  EXPECT_CALL(mock_buzzer_, MakeBuzz("x"))
      .WillRepeatedly([](StringPiece text) {
        return MakeUnique<Buzz>(AccessLevel::kInternal);
      });

  EXPECT_NE(nullptr, mock_buzzer_.MakeBuzz("x"));
  EXPECT_NE(nullptr, mock_buzzer_.MakeBuzz("x"));
```

Every time this `EXPECT_CALL` fires, a new `unique_ptr<Buzz>` will be created and returned. You cannot do this with `Return(ByMove(...))`.

That covers returning move-only values; but how do we work with methods accepting move-only arguments? The answer is that they work normally, although some actions will not compile when any of method's arguments are move-only. You can always use `Return`, or a lambda or functor:

```
  using ::testing::Unused;

  EXPECT_CALL(mock_buzzer_, ShareBuzz(NotNull(), _)).WillOnce(Return(true));
  EXPECT_TRUE(mock_buzzer_.ShareBuzz(MakeUnique<Buzz>(AccessLevel::kInternal)),
              0);

  EXPECT_CALL(mock_buzzer_, ShareBuzz(_, _)).WillOnce(
      [](std::unique_ptr<Buzz> buzz, Unused) { return buzz != nullptr; });
  EXPECT_FALSE(mock_buzzer_.ShareBuzz(nullptr, 0));
```

Many built-in actions (`WithArgs`, `WithoutArgs`,`DeleteArg`, `SaveArg`, ...) could in principle support move-only arguments, but the support for this is not implemented yet. If this is blocking you, please file a bug.

A few actions (e.g. `DoAll`) copy their arguments internally, so they can never work with non-copyable objects; you'll have to use functors instead.

Legacy workarounds for move-only types {#LegacyMoveOnly}

Support for move-only function arguments was only introduced to gMock in April of 2017. In older code, you may encounter the following workaround for the lack of this feature (it is no longer necessary - we're including it just for reference):

```
class MockBuzzer : public Buzzer {
 public:
  MOCK_METHOD(bool, DoShareBuzz, (Buzz* buzz, Time timestamp));
  bool ShareBuzz(std::unique_ptr<Buzz> buzz, Time timestamp) override {
    return DoShareBuzz(buzz.get(), timestamp);
  }
};
```

The trick is to delegate the `ShareBuzz()` method to a mock method (let’s call it `DoShareBuzz()`) that does not take move-only parameters. Then, instead of setting expectations on `ShareBuzz()`, you set them on the `DoShareBuzz()` mock method:

```
  MockBuzzer mock_buzzer_;
  EXPECT_CALL(mock_buzzer_, DoShareBuzz(NotNull(), _));

  // When one calls ShareBuzz() on the MockBuzzer like this, the call is
  // forwarded to DoShareBuzz(), which is mocked.  Therefore this statement
  // will trigger the above EXPECT_CALL.
  mock_buzzer_.ShareBuzz(MakeUnique<Buzz>(AccessLevel::kInternal), 0);
```

Making the Compilation Faster

Believe it or not, the *vast majority* of the time spent on compiling a mock class is in generating its constructor and destructor, as they perform non-trivial tasks (e.g. verification of the expectations). What's more, mock methods with different signatures have different types and thus their constructors/destructors need to be generated by the compiler separately. As a result, if you mock many different types of methods, compiling your mock class can get really slow.

If you are experiencing slow compilation, you can move the definition of your mock class' constructor and destructor out of the class body and into a `.cc` file. This way, even if you `#include` your mock class in N files, the compiler only needs to generate its constructor and destructor once, resulting in a much faster compilation.

Let's illustrate the idea using an example. Here's the definition of a mock class before applying this recipe:

```
// File mock_foo.h.
...
class MockFoo : public Foo {
 public:
  // Since we don't declare the constructor or the destructor,
  // the compiler will generate them in every translation unit
  // where this mock class is used.

  MOCK_METHOD(int, DoThis, (), (override));
  MOCK_METHOD(bool, DoThat, (const char* str), (override));
  ... more mock methods ...
};
```

After the change, it would look like:

```
// File mock_foo.h.
...
class MockFoo : public Foo {
 public:
  // The constructor and destructor are declared, but not defined, here.
  MockFoo();
  virtual ~MockFoo();

  MOCK_METHOD(int, DoThis, (), (override));
  MOCK_METHOD(bool, DoThat, (const char* str), (override));
  ... more mock methods ...
};
```

and

```
// File mock_foo.cc.
#include "path/to/mock_foo.h"

// The definitions may appear trivial, but the functions actually do a
// lot of things through the constructors/destructors of the member
// variables used to implement the mock methods.
MockFoo::MockFoo() {}
MockFoo::~MockFoo() {}
```

Forcing a Verification

When it's being destroyed, your friendly mock object will automatically verify that all expectations on it have been satisfied, and will generate googletest failures if not. This is convenient as it leaves you with one less thing to worry about. That is, unless you are not sure if your mock object will be destroyed.

How could it be that your mock object won't eventually be destroyed? Well, it might be created on the heap and owned by the code you are testing. Suppose there's a bug in that code and it doesn't delete the mock object properly - you could end up with a passing test when there's actually a bug.

Using a heap checker is a good idea and can alleviate the concern, but its implementation is not 100% reliable. So, sometimes you do want to *force* gMock to verify a mock object before it is (hopefully) destructed. You can do this with `Mock::VerifyAndClearExpectations(&mock_object)`:

```
TEST(MyServerTest, ProcessesRequest) {
  using ::testing::Mock;

  MockFoo* const foo = new MockFoo;
  EXPECT_CALL(*foo, ...)...;
  // ... other expectations ...

  // server now owns foo.
  MyServer server(foo);
  server.ProcessRequest(...);

  // In case that server's destructor will forget to delete foo,
  // this will verify the expectations anyway.
  Mock::VerifyAndClearExpectations(foo);
}  // server is destroyed when it goes out of scope here.
```

{: .callout .tip}
Tip: The `Mock::VerifyAndClearExpectations()` function returns a `bool` to indicate whether the verification was successful (`true` for yes), so you can wrap that function call inside a `ASSERT_TRUE()` if there is no point going further when the verification has failed.

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.

Using Checkpoints {#UsingCheckPoints}

Sometimes you might want to test a mock object's behavior in phases whose sizes are each manageable, or you might want to set more detailed expectations about which API calls invoke which mock functions.

A technique you can use is to put the expectations in a sequence and insert calls to a dummy "checkpoint" function at specific places. Then you can verify that the mock function calls do happen at the right time. For example, if you are exercising the code:

```
  Foo(1);
  Foo(2);
  Foo(3);
```

and want to verify that `Foo(1)` and `Foo(3)` both invoke `mock.Bar("a")`, but `Foo(2)` doesn't invoke anything, you can write:

```
using ::testing::MockFunction;

TEST(FooTest, InvokesBarCorrectly) {
  MyMock mock;
  // Class MockFunction<F> has exactly one mock method.  It is named
  // Call() and has type F.
  MockFunction<void(string check_point_name)> check;
  {
    InSequence s;

    EXPECT_CALL(mock, Bar("a"));
    EXPECT_CALL(check, Call("1"));
    EXPECT_CALL(check, Call("2"));
    EXPECT_CALL(mock, Bar("a"));
  }
  Foo(1);
  check.Call("1");
  Foo(2);
  check.Call("2");
  Foo(3);
}
```

The expectation spec says that the first `Bar("a")` call must happen before checkpoint "1", the second `Bar("a")` call must happen after checkpoint "2", and nothing should happen between the two checkpoints. The explicit checkpoints make it clear which `Bar("a")` is called by which call to `Foo()`.

Mocking Destructors

Sometimes you want to make sure a mock object is destructed at the right time, e.g. after `bar->A()` is called but before `bar->B()` is called. We already know that you can specify constraints on the order of mock function calls, so all we need to do is to mock the destructor of the mock function.

This sounds simple, except for one problem: a destructor is a special function with special syntax and special semantics, and the `MOCK_METHOD` macro doesn't work for it:

```
MOCK_METHOD(void, ~MockFoo, ());  // Won't compile!
```

The good news is that you can use a simple pattern to achieve the same effect. First, add a mock function `Die()` to your mock class and call it in the destructor, like this:

```
class MockFoo : public Foo {
  ...
  // Add the following two lines to the mock class.
  MOCK_METHOD(void, Die, ());
  ~MockFoo() override { Die(); }
};
```

(If the name `Die()` clashes with an existing symbol, choose another name.) Now, we have translated the problem of testing when a `MockFoo` object dies to testing when its `Die()` method is called:

```
  MockFoo* foo = new MockFoo;
  MockBar* bar = new MockBar;
  ...
  {
    InSequence s;

    // Expects *foo to die after bar->A() and before bar->B().
    EXPECT_CALL(*bar, A());
    EXPECT_CALL(*foo, Die());
    EXPECT_CALL(*bar, B());
  }
```

And that's that.

Using gMock and Threads {#UsingThreads}

In a **unit** test, it's best if you could isolate and test a piece of code in a single-threaded context. That avoids race conditions and dead locks, and makes debugging your test much easier.

Yet most programs are multi-threaded, and sometimes to test something we need to pound on it from more than one thread. gMock works for this purpose too.

Remember the steps for using a mock:

1. Create a mock object `foo`.
2. Set its default actions and expectations using `ON_CALL()` and `EXPECT_CALL()`.
3. The code under test calls methods of `foo`.
4. Optionally, verify and reset the mock.
5. Destroy the mock yourself, or let the code under test destroy it. The destructor will automatically verify it.

If you follow the following simple rules, your mocks and threads can live happily together:

- Execute your *test code* (as opposed to the code being tested) in *one* thread. This makes your test easy to follow.
- Obviously, you can do step #1 without locking.
- When doing step #2 and #5, make sure no other thread is accessing `foo`. Obvious too, huh?
- #3 and #4 can be done either in one thread or in multiple threads - anyway you want. gMock takes care of the locking, so you don't have to do any - unless required by your test logic.

If you violate the rules (for example, if you set expectations on a mock while another thread is calling its methods), you get undefined behavior. That's not fun, so don't do it.

gMock guarantees that the action for a mock function is done in the same thread that called the mock function. For example, in

```
  EXPECT_CALL(mock, Foo(1))
      .WillOnce(action1);
  EXPECT_CALL(mock, Foo(2))
      .WillOnce(action2);
```

if `Foo(1)` is called in thread 1 and `Foo(2)` is called in thread 2, gMock will execute `action1` in thread 1 and `action2` in thread 2.

gMock does *not* impose a sequence on actions performed in different threads (doing so may create deadlocks as the actions may need to cooperate). This means that the execution of `action1` and `action2` in the above example *may* interleave. If this is a problem, you should add proper synchronization logic to `action1` and `action2` to make the test thread-safe.

Also, remember that `DefaultValue<T>` is a global resource that potentially affects *all* living mock objects in your program. Naturally, you won't want to mess with it from multiple threads or when there still are mocks in action.

Controlling How Much Information gMock Prints

When gMock sees something that has the potential of being an error (e.g. a mock function with no expectation is called, a.k.a. an uninteresting call, which is allowed but perhaps you forgot to explicitly ban the call), it prints some warning messages, including the arguments of the function, the return value, and the stack trace. Hopefully this will remind you to take a look and see if there is indeed a problem.

Sometimes you are confident that your tests are correct and may not appreciate such friendly messages. Some other times, you are debugging your tests or learning about the behavior of the code you are testing, and wish you could observe every mock call that happens (including argument values, the return value, and the stack trace). Clearly, one size doesn't fit all.

You can control how much gMock tells you using the `--gmock_verbose=LEVEL` command-line flag, where `LEVEL` is a string with three possible values:

- `info`: gMock will print all informational messages, warnings, and errors (most verbose). At this setting, gMock will also log any calls to the `ON_CALL/EXPECT_CALL` macros. It will include a stack trace in "uninteresting call" warnings.
- `warning`: gMock will print both warnings and errors (less verbose); it will omit the stack traces in "uninteresting call" warnings. This is the default.
- `error`: gMock will print errors only (least verbose).

Alternatively, you can adjust the value of that flag from within your tests like so:

```
  ::testing::FLAGS_gmock_verbose = "error";
```

If you find gMock printing too many stack frames with its informational or warning messages, remember that you can control their amount with the `--gtest_stack_trace_depth=max_depth` flag.

Now, judiciously use the right flag to enable gMock serve you better!

Gaining Super Vision into Mock Calls

You have a test using gMock. It fails: gMock tells you some expectations aren't satisfied. However, you aren't sure why: Is there a typo somewhere in the matchers? Did you mess up the order of the `EXPECT_CALL`s? Or is the code under test doing something wrong? How can you find out the cause?

Won't it be nice if you have X-ray vision and can actually see the trace of all `EXPECT_CALL`s and mock method calls as they are made? For each call, would you like to see its actual argument values and which `EXPECT_CALL` gMock thinks it matches? If you still need some help to figure out who made these calls, how about being able to see the complete stack trace at each mock call?

You can unlock this power by running your test with the `--gmock_verbose=info` flag. For example, given the test program:

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

using testing::_;
using testing::HasSubstr;
using testing::Return;

class MockFoo {
 public:
  MOCK_METHOD(void, F, (const string& x, const string& y));
};

TEST(Foo, Bar) {
  MockFoo mock;
  EXPECT_CALL(mock, F(_, _)).WillRepeatedly(Return());
  EXPECT_CALL(mock, F("a", "b"));
  EXPECT_CALL(mock, F("c", HasSubstr("d")));

  mock.F("a", "good");
  mock.F("a", "b");
}
```

if you run it with `--gmock_verbose=info`, you will see this output:

```
[ RUN       ] Foo.Bar

foo_test.cc:14: EXPECT_CALL(mock, F(_, _)) invoked
Stack trace: ...

foo_test.cc:15: EXPECT_CALL(mock, F("a", "b")) invoked
Stack trace: ...

foo_test.cc:16: EXPECT_CALL(mock, F("c", HasSubstr("d"))) invoked
Stack trace: ...

foo_test.cc:14: Mock function call matches EXPECT_CALL(mock, F(_, _))...
    Function call: F(@0x7fff7c8dad40"a",@0x7fff7c8dad10"good")
Stack trace: ...

foo_test.cc:15: Mock function call matches EXPECT_CALL(mock, F("a", "b"))...
    Function call: F(@0x7fff7c8dada0"a",@0x7fff7c8dad70"b")
Stack trace: ...

foo_test.cc:16: Failure
Actual function call count doesn't match EXPECT_CALL(mock, F("c", HasSubstr("d")))...
         Expected: to be called once
           Actual: never called - unsatisfied and active
[  FAILED  ] Foo.Bar
```

Suppose the bug is that the `"c"` in the third `EXPECT_CALL` is a typo and should actually be `"a"`. With the above message, you should see that the actual `F("a", "good")` call is matched by the first `EXPECT_CALL`, not the third as you thought. From that it should be obvious that the third `EXPECT_CALL` is written wrong. Case solved.

If you are interested in the mock call trace but not the stack traces, you can combine `--gmock_verbose=info` with `--gtest_stack_trace_depth=0` on the test command line.

Running Tests in Emacs

If you build and run your tests in Emacs using the `M-x google-compile` command (as many googletest users do), the source file locations of gMock and googletest errors will be highlighted. Just press `<Enter>` on one of them and you'll be taken to the offending line. Or, you can just type `C-x`` to jump to the next error.

To make it even easier, you can add the following lines to your `~/.emacs` file:

```
(global-set-key "\M-m"  'google-compile)  ; m is for make
(global-set-key [M-down] 'next-error)
(global-set-key [M-up]  '(lambda () (interactive) (next-error -1)))
```

Then you can type `M-m` to start a build (if you want to run the test as well, just make sure `foo_test.run` or `runtests` is in the build command you supply after typing `M-m`), or `M-up`/`M-down` to move back and forth between errors.

Extending gMock

Writing New Matchers Quickly {#NewMatchers}

{: .callout .warning}
WARNING: gMock does not guarantee when or how many times a matcher will be invoked. Therefore, all matchers must be functionally pure. See this section for more details.

The `MATCHER*` family of macros can be used to define custom matchers easily. The syntax:

```
MATCHER(name, description_string_expression) { statements; }
```

will define a matcher with the given name that executes the statements, which must return a `bool` to indicate if the match succeeds. Inside the statements, you can refer to the value being matched by `arg`, and refer to its type by `arg_type`.

The *description string* is a `string`-typed expression that documents what the matcher does, and is used to generate the failure message when the match fails. It can (and should) reference the special `bool` variable `negation`, and should evaluate to the description of the matcher when `negation` is `false`, or that of the matcher's negation when `negation` is `true`.

For convenience, we allow the description string to be empty (`""`), in which case gMock will use the sequence of words in the matcher name as the description.

For example:

```
MATCHER(IsDivisibleBy7, "") { return (arg % 7) == 0; }
```

allows you to write

```
  // Expects mock_foo.Bar(n) to be called where n is divisible by 7.
  EXPECT_CALL(mock_foo, Bar(IsDivisibleBy7()));
```

or,

```
  using ::testing::Not;
  ...
  // Verifies that a value is divisible by 7 and the other is not.
  EXPECT_THAT(some_expression, IsDivisibleBy7());
  EXPECT_THAT(some_other_expression, Not(IsDivisibleBy7()));
```

If the above assertions fail, they will print something like:

```
  Value of: some_expression
  Expected: is divisible by 7
    Actual: 27
  ...
  Value of: some_other_expression
  Expected: not (is divisible by 7)
    Actual: 21
```

where the descriptions `"is divisible by 7"` and `"not (is divisible by 7)"` are automatically calculated from the matcher name `IsDivisibleBy7`.

As you may have noticed, the auto-generated descriptions (especially those for the negation) may not be so great. You can always override them with a `string` expression of your own:

```
MATCHER(IsDivisibleBy7,
        absl::StrCat(negation ? "isn't" : "is", " divisible by 7")) {
  return (arg % 7) == 0;
}
```

Optionally, you can stream additional information to a hidden argument named `result_listener` to explain the match result. For example, a better definition of `IsDivisibleBy7` is:

```
MATCHER(IsDivisibleBy7, "") {
  if ((arg % 7) == 0)
    return true;

  *result_listener << "the remainder is " << (arg % 7);
  return false;
}
```

With this definition, the above assertion will give a better message:

```
  Value of: some_expression
  Expected: is divisible by 7
    Actual: 27 (the remainder is 6)
```

You should let `MatchAndExplain()` print *any additional information* that can help a user understand the match result. Note that it should explain why the match succeeds in case of a success (unless it's obvious) - this is useful when the matcher is used inside `Not()`. There is no need to print the argument value itself, as gMock already prints it for you.

{: .callout .note}
NOTE: The type of the value being matched (`arg_type`) is determined by the context in which you use the matcher and is supplied to you by the compiler, so you don't need to worry about declaring it (nor can you). This allows the matcher to be polymorphic. For example, `IsDivisibleBy7()` can be used to match any type where the value of `(arg % 7) == 0` can be implicitly converted to a `bool`. In the `Bar(IsDivisibleBy7())` example above, if method `Bar()` takes an `int`, `arg_type` will be `int`; if it takes an `unsigned long`, `arg_type` will be `unsigned long`; and so on.

Writing New Parameterized Matchers Quickly

Sometimes you'll want to define a matcher that has parameters. For that you can use the macro:

```
MATCHER_P(name, param_name, description_string) { statements; }
```

where the description string can be either `""` or a `string` expression that references `negation` and `param_name`.

For example:

```
MATCHER_P(HasAbsoluteValue, value, "") { return abs(arg) == value; }
```

will allow you to write:

```
  EXPECT_THAT(Blah("a"), HasAbsoluteValue(n));
```

which may lead to this message (assuming `n` is 10):

```
  Value of: Blah("a")
  Expected: has absolute value 10
    Actual: -9
```

Note that both the matcher description and its parameter are printed, making the message human-friendly.

In the matcher definition body, you can write `foo_type` to reference the type of a parameter named `foo`. For example, in the body of `MATCHER_P(HasAbsoluteValue, value)` above, you can write `value_type` to refer to the type of `value`.

gMock also provides `MATCHER_P2`, `MATCHER_P3`, ..., up to `MATCHER_P10` to support multi-parameter matchers:

```
MATCHER_Pk(name, param_1, ..., param_k, description_string) { statements; }
```

Please note that the custom description string is for a particular *instance* of the matcher, where the parameters have been bound to actual values. Therefore usually you'll want the parameter values to be part of the description. gMock lets you do that by referencing the matcher parameters in the description string expression.

For example,

```
using ::testing::PrintToString;
MATCHER_P2(InClosedRange, low, hi,
           absl::StrFormat("%s in range [%s, %s]", negation ? "isn't" : "is",
                           PrintToString(low), PrintToString(hi))) {
  return low <= arg && arg <= hi;
}
...
EXPECT_THAT(3, InClosedRange(4, 6));
```

would generate a failure that contains the message:

```
  Expected: is in range [4, 6]
```

If you specify `""` as the description, the failure message will contain the sequence of words in the matcher name followed by the parameter values printed as a tuple. For example,

```
  MATCHER_P2(InClosedRange, low, hi, "") { ... }
  ...
  EXPECT_THAT(3, InClosedRange(4, 6));
```

would generate a failure that contains the text:

```
  Expected: in closed range (4, 6)
```

For the purpose of typing, you can view

```
MATCHER_Pk(Foo, p1, ..., pk, description_string) { ... }
```

as shorthand for

```
template <typename p1_type, ..., typename pk_type>
FooMatcherPk<p1_type, ..., pk_type>
Foo(p1_type p1, ..., pk_type pk) { ... }
```

When you write `Foo(v1, ..., vk)`, the compiler infers the types of the parameters `v1`, ..., and `vk` for you. If you are not happy with the result of the type inference, you can specify the types by explicitly instantiating the template, as in `Foo<long, bool>(5, false)`. As said earlier, you don't get to (or need to) specify `arg_type` as that's determined by the context in which the matcher is used.

You can assign the result of expression `Foo(p1, ..., pk)` to a variable of type `FooMatcherPk<p1_type, ..., pk_type>`. This can be useful when composing matchers. Matchers that don't have a parameter or have only one parameter have special types: you can assign `Foo()` to a `FooMatcher`-typed variable, and assign `Foo(p)` to a `FooMatcherP<p_type>`-typed variable.

While you can instantiate a matcher template with reference types, passing the parameters by pointer usually makes your code more readable. If, however, you still want to pass a parameter by reference, be aware that in the failure message generated by the matcher you will see the value of the referenced object but not its address.

You can overload matchers with different numbers of parameters:

```
MATCHER_P(Blah, a, description_string_1) { ... }
MATCHER_P2(Blah, a, b, description_string_2) { ... }
```

While it's tempting to always use the `MATCHER*` macros when defining a new matcher, you should also consider implementing the matcher interface directly instead (see the recipes that follow), especially if you need to use the matcher a lot. While these approaches require more work, they give you more control on the types of the value being matched and the matcher parameters, which in general leads to better compiler error messages that pay off in the long run. They also allow overloading matchers based on parameter types (as opposed to just based on the number of parameters).

Writing New Monomorphic Matchers

A matcher of argument type `T` implements the matcher interface for `T` and does two things: it tests whether a value of type `T` matches the matcher, and can describe what kind of values it matches. The latter ability is used for generating readable error messages when expectations are violated.

A matcher of `T` must declare a typedef like:

```
using is_gtest_matcher = void;
```

and supports the following operations:

```
// Match a value and optionally explain into an ostream.
bool matched = matcher.MatchAndExplain(value, maybe_os);
// where `value` is of type `T` and
// `maybe_os` is of type `std::ostream*`, where it can be null if the caller
// is not interested in there textual explanation.

matcher.DescribeTo(os);
matcher.DescribeNegationTo(os);
// where `os` is of type `std::ostream*`.
```

If you need a custom matcher but `Truly()` is not a good option (for example, you may not be happy with the way `Truly(predicate)` describes itself, or you may want your matcher to be polymorphic as `Eq(value)` is), you can define a matcher to do whatever you want in two steps: first implement the matcher interface, and then define a factory function to create a matcher instance. The second step is not strictly needed but it makes the syntax of using the matcher nicer.

For example, you can define a matcher to test whether an `int` is divisible by 7 and then use it like this:

```
using ::testing::Matcher;

class DivisibleBy7Matcher {
 public:
  using is_gtest_matcher = void;

  bool MatchAndExplain(int n, std::ostream*) const {
    return (n % 7) == 0;
  }

  void DescribeTo(std::ostream* os) const {
    *os << "is divisible by 7";
  }

  void DescribeNegationTo(std::ostream* os) const {
    *os << "is not divisible by 7";
  }
};

Matcher<int> DivisibleBy7() {
  return DivisibleBy7Matcher();
}

...
  EXPECT_CALL(foo, Bar(DivisibleBy7()));
```

You may improve the matcher message by streaming additional information to the `os` argument in `MatchAndExplain()`:

```
class DivisibleBy7Matcher {
 public:
  bool MatchAndExplain(int n, std::ostream* os) const {
    const int remainder = n % 7;
    if (remainder != 0 && os != nullptr) {
      *os << "the remainder is " << remainder;
    }
    return remainder == 0;
  }
  ...
};
```

Then, `EXPECT_THAT(x, DivisibleBy7());` may generate a message like this:

```
Value of: x
Expected: is divisible by 7
  Actual: 23 (the remainder is 2)
```

{: .callout .tip}
Tip: for convenience, `MatchAndExplain()` can take a `MatchResultListener*` instead of `std::ostream*`.

Writing New Polymorphic Matchers

Expanding what we learned above to *polymorphic* matchers is now just as simple as adding templates in the right place.

```
class NotNullMatcher {
 public:
  using is_gtest_matcher = void;

  // To implement a polymorphic matcher, we just need to make MatchAndExplain a
  // template on its first argument.

  // In this example, we want to use NotNull() with any pointer, so
  // MatchAndExplain() accepts a pointer of any type as its first argument.
  // In general, you can define MatchAndExplain() as an ordinary method or
  // a method template, or even overload it.
  template <typename T>
  bool MatchAndExplain(T* p, std::ostream*) const {
    return p != nullptr;
  }

  // Describes the property of a value matching this matcher.
  void DescribeTo(std::ostream* os) const { *os << "is not NULL"; }

  // Describes the property of a value NOT matching this matcher.
  void DescribeNegationTo(std::ostream* os) const { *os << "is NULL"; }
};

NotNullMatcher NotNull() {
  return NotNullMatcher();
}

...

  EXPECT_CALL(foo, Bar(NotNull()));  // The argument must be a non-NULL pointer.
```

Legacy Matcher Implementation

Defining matchers used to be somewhat more complicated, in which it required several supporting classes and virtual functions. To implement a matcher for type `T` using the legacy API you have to derive from `MatcherInterface<T>` and call `MakeMatcher` to construct the object.

The interface looks like this:

```
class MatchResultListener {
 public:
  ...
  // Streams x to the underlying ostream; does nothing if the ostream
  // is NULL.
  template <typename T>
  MatchResultListener& operator<<(const T& x);
  // Returns the underlying ostream.
  std::ostream* stream();
};

template <typename T>
class MatcherInterface {
 public:
  virtual ~MatcherInterface();

  // Returns true if and only if the matcher matches x; also explains the match
  // result to 'listener'.
  virtual bool MatchAndExplain(T x, MatchResultListener* listener) const = 0;

  // Describes this matcher to an ostream.
  virtual void DescribeTo(std::ostream* os) const = 0;

  // Describes the negation of this matcher to an ostream.
  virtual void DescribeNegationTo(std::ostream* os) const;
};
```

Fortunately, most of the time you can define a polymorphic matcher easily with the help of `MakePolymorphicMatcher()`. Here's how you can define `NotNull()` as an example:

```
using ::testing::MakePolymorphicMatcher;
using ::testing::MatchResultListener;
using ::testing::PolymorphicMatcher;

class NotNullMatcher {
 public:
  // To implement a polymorphic matcher, first define a COPYABLE class
  // that has three members MatchAndExplain(), DescribeTo(), and
  // DescribeNegationTo(), like the following.

  // In this example, we want to use NotNull() with any pointer, so
  // MatchAndExplain() accepts a pointer of any type as its first argument.
  // In general, you can define MatchAndExplain() as an ordinary method or
  // a method template, or even overload it.
  template <typename T>
  bool MatchAndExplain(T* p,
                       MatchResultListener* /* listener */) const {
    return p != NULL;
  }

  // Describes the property of a value matching this matcher.
  void DescribeTo(std::ostream* os) const { *os << "is not NULL"; }

  // Describes the property of a value NOT matching this matcher.
  void DescribeNegationTo(std::ostream* os) const { *os << "is NULL"; }
};

// To construct a polymorphic matcher, pass an instance of the class
// to MakePolymorphicMatcher().  Note the return type.
PolymorphicMatcher<NotNullMatcher> NotNull() {
  return MakePolymorphicMatcher(NotNullMatcher());
}

...

  EXPECT_CALL(foo, Bar(NotNull()));  // The argument must be a non-NULL pointer.
```

{: .callout .note}
Note: Your polymorphic matcher class does **not** need to inherit from `MatcherInterface` or any other class, and its methods do **not** need to be virtual.

Like in a monomorphic matcher, you may explain the match result by streaming additional information to the `listener` argument in `MatchAndExplain()`.

Writing New Cardinalities

A cardinality is used in `Times()` to tell gMock how many times you expect a call to occur. It doesn't have to be exact. For example, you can say `AtLeast(5)` or `Between(2, 4)`.

If the built-in set of cardinalities doesn't suit you, you are free to define your own by implementing the following interface (in namespace `testing`):

```
class CardinalityInterface {
 public:
  virtual ~CardinalityInterface();

  // Returns true if and only if call_count calls will satisfy this cardinality.
  virtual bool IsSatisfiedByCallCount(int call_count) const = 0;

  // Returns true if and only if call_count calls will saturate this
  // cardinality.
  virtual bool IsSaturatedByCallCount(int call_count) const = 0;

  // Describes self to an ostream.
  virtual void DescribeTo(std::ostream* os) const = 0;
};
```

For example, to specify that a call must occur even number of times, you can write

```
using ::testing::Cardinality;
using ::testing::CardinalityInterface;
using ::testing::MakeCardinality;

class EvenNumberCardinality : public CardinalityInterface {
 public:
  bool IsSatisfiedByCallCount(int call_count) const override {
    return (call_count % 2) == 0;
  }

  bool IsSaturatedByCallCount(int call_count) const override {
    return false;
  }

  void DescribeTo(std::ostream* os) const {
    *os << "called even number of times";
  }
};

Cardinality EvenNumber() {
  return MakeCardinality(new EvenNumberCardinality);
}

...
  EXPECT_CALL(foo, Bar(3))
      .Times(EvenNumber());
```

Writing New Actions {#QuickNewActions}

If the built-in actions don't work for you, you can easily define your own one. All you need is a call operator with a signature compatible with the mocked function. So you can use a lambda:

```
MockFunction<int(int)> mock;
EXPECT_CALL(mock, Call).WillOnce([](const int input) { return input * 7; });
EXPECT_EQ(14, mock.AsStdFunction()(2));
```

Or a struct with a call operator (even a templated one):

```
struct MultiplyBy {
  template <typename T>
  T operator()(T arg) { return arg * multiplier; }

  int multiplier;
};

// Then use:
// EXPECT_CALL(...).WillOnce(MultiplyBy{7});
```

It's also fine for the callable to take no arguments, ignoring the arguments supplied to the mock function:

```
MockFunction<int(int)> mock;
EXPECT_CALL(mock, Call).WillOnce([] { return 17; });
EXPECT_EQ(17, mock.AsStdFunction()(0));
```

When used with `WillOnce`, the callable can assume it will be called at most once and is allowed to be a move-only type:

```
// An action that contains move-only types and has an &&-qualified operator,
// demanding in the type system that it be called at most once. This can be
// used with WillOnce, but the compiler will reject it if handed to
// WillRepeatedly.
struct MoveOnlyAction {
  std::unique_ptr<int> move_only_state;
  std::unique_ptr<int> operator()() && { return std::move(move_only_state); }
};

MockFunction<std::unique_ptr<int>()> mock;
EXPECT_CALL(mock, Call).WillOnce(MoveOnlyAction{std::make_unique<int>(17)});
EXPECT_THAT(mock.AsStdFunction()(), Pointee(Eq(17)));
```

More generally, to use with a mock function whose signature is `R(Args...)` the object can be anything convertible to `OnceAction<R(Args...)>` or `Action<R(Args...)`>. The difference between the two is that `OnceAction` has weaker requirements (`Action` requires a copy-constructible input that can be called repeatedly whereas `OnceAction` requires only move-constructible and supports `&&`-qualified call operators), but can be used only with `WillOnce`. `OnceAction` is typically relevant only when supporting move-only types or actions that want a type-system guarantee that they will be called at most once.

Typically the `OnceAction` and `Action` templates need not be referenced directly in your actions: a struct or class with a call operator is sufficient, as in the examples above. But fancier polymorphic actions that need to know the specific return type of the mock function can define templated conversion operators to make that possible. See `gmock-actions.h` for examples.

Legacy macro-based Actions

Before C++11, the functor-based actions were not supported; the old way of writing actions was through a set of `ACTION*` macros. We suggest to avoid them in new code; they hide a lot of logic behind the macro, potentially leading to harder-to-understand compiler errors. Nevertheless, we cover them here for completeness.

By writing

```
ACTION(name) { statements; }
```

in a namespace scope (i.e. not inside a class or function), you will define an action with the given name that executes the statements. The value returned by `statements` will be used as the return value of the action. Inside the statements, you can refer to the K-th (0-based) argument of the mock function as `argK`. For example:

```
ACTION(IncrementArg1) { return ++(*arg1); }
```

allows you to write

```
... WillOnce(IncrementArg1());
```

Note that you don't need to specify the types of the mock function arguments. Rest assured that your code is type-safe though: you'll get a compiler error if `*arg1` doesn't support the `++` operator, or if the type of `++(*arg1)` isn't compatible with the mock function's return type.

Another example:

```
ACTION(Foo) {
  (*arg2)(5);
  Blah();
  *arg1 = 0;
  return arg0;
}
```

defines an action `Foo()` that invokes argument #2 (a function pointer) with 5, calls function `Blah()`, sets the value pointed to by argument #1 to 0, and returns argument #0.

For more convenience and flexibility, you can also use the following pre-defined symbols in the body of `ACTION`:

`argK_type`	**The type of the K-th (0-based) argument of the mock function**
`args`	All arguments of the mock function as a tuple
`args_type`	The type of all arguments of the mock function as a tuple
`return_type`	The return type of the mock function
`function_type`	The type of the mock function

For example, when using an `ACTION` as a stub action for mock function:

```
int DoSomething(bool flag, int* ptr);
```

we have:

Pre-defined Symbol	Is Bound To
`arg0`	the value of `flag`
`arg0_type`	the type `bool`
`arg1`	the value of `ptr`
`arg1_type`	the type `int*`
`args`	the tuple `(flag, ptr)`
`args_type`	the type `std::tuple<bool, int*>`
`return_type`	the type `int`
`function_type`	the type `int(bool, int*)`

Legacy macro-based parameterized Actions

Sometimes you'll want to parameterize an action you define. For that we have another macro

```
ACTION_P(name, param) { statements; }
```

For example,

```
ACTION_P(Add, n) { return arg0 + n; }
```

will allow you to write

```
// Returns argument #0 + 5.
... WillOnce(Add(5));
```

For convenience, we use the term *arguments* for the values used to invoke the mock function, and the term *parameters* for the values used to instantiate an action.

Note that you don't need to provide the type of the parameter either. Suppose the parameter is named `param`, you can also use the gMock-defined symbol `param_type` to refer to the type of the parameter as inferred by the compiler. For example, in the body of `ACTION_P(Add, n)` above, you can write `n_type` for the type of `n`.

gMock also provides `ACTION_P2`, `ACTION_P3`, and etc to support multi-parameter actions. For example,

```
ACTION_P2(ReturnDistanceTo, x, y) {
  double dx = arg0 - x;
  double dy = arg1 - y;
  return sqrt(dx*dx + dy*dy);
}
```

lets you write

```
... WillOnce(ReturnDistanceTo(5.0, 26.5));
```

You can view `ACTION` as a degenerated parameterized action where the number of parameters is 0.

You can also easily define actions overloaded on the number of parameters:

```
ACTION_P(Plus, a) { ... }
ACTION_P2(Plus, a, b) { ... }
```

Restricting the Type of an Argument or Parameter in an ACTION

For maximum brevity and reusability, the `ACTION*` macros don't ask you to provide the types of the mock function arguments and the action parameters. Instead, we let the compiler infer the types for us.

Sometimes, however, we may want to be more explicit about the types. There are several tricks to do that. For example:

```
ACTION(Foo) {
  // Makes sure arg0 can be converted to int.
  int n = arg0;
  ... use n instead of arg0 here ...
}

ACTION_P(Bar, param) {
  // Makes sure the type of arg1 is const char*.
  ::testing::StaticAssertTypeEq<const char*, arg1_type>();

  // Makes sure param can be converted to bool.
  bool flag = param;
}
```

where `StaticAssertTypeEq` is a compile-time assertion in googletest that verifies two types are the same.

Writing New Action Templates Quickly

Sometimes you want to give an action explicit template parameters that cannot be inferred from its value parameters. `ACTION_TEMPLATE()` supports that and can be viewed as an extension to `ACTION()` and `ACTION_P*()`.

The syntax:

```
ACTION_TEMPLATE(ActionName,
                HAS_m_TEMPLATE_PARAMS(kind1, name1, ..., kind_m, name_m),
                AND_n_VALUE_PARAMS(p1, ..., p_n)) { statements; }
```

defines an action template that takes *m* explicit template parameters and *n* value parameters, where *m* is in [1, 10] and *n* is in [0, 10]. `name_i` is the name of the *i*-th template parameter, and `kind_i` specifies whether it's a `typename`, an integral constant, or a template. `p_i` is the name of the *i*-th value parameter.

Example:

```
// DuplicateArg<k, T>(output) converts the k-th argument of the mock
// function to type T and copies it to *output.
ACTION_TEMPLATE(DuplicateArg,
                // Note the comma between int and k:
                HAS_2_TEMPLATE_PARAMS(int, k, typename, T),
                AND_1_VALUE_PARAMS(output)) {
  *output = T(std::get<k>(args));
}
```

To create an instance of an action template, write:

```
ActionName<t1, ..., t_m>(v1, ..., v_n)
```

where the `t` s are the template arguments and the `v` s are the value arguments. The value argument types are inferred by the compiler. For example:

```
using ::testing::_;
...
  int n;
  EXPECT_CALL(mock, Foo).WillOnce(DuplicateArg<1, unsigned char>(&n));
```

If you want to explicitly specify the value argument types, you can provide additional template arguments:

```
ActionName<t1, ..., t_m, u1, ..., u_k>(v1, ..., v_n)
```

where `u_i` is the desired type of `v_i`.

`ACTION_TEMPLATE` and `ACTION` / `ACTION_P*` can be overloaded on the number of value parameters, but not on the number of template parameters. Without the restriction, the meaning of the following is unclear:

```
  OverloadedAction<int, bool>(x);
```

Are we using a single-template-parameter action where `bool` refers to the type of `x`, or a two-template-action where the compiler is asked to infer the type of `x`?

Using the ACTION Object's Type

If you are writing a function that returns an `ACTION` object, you'll need to know its type. The type depends on the macro used to define the action and the parameter types. The rule is relatively simple:

Given Definition	Expression	Has Type
`ACTION(Foo)`	`Foo()`	`FooAction`
`ACTION_TEMPLATE(Foo, HAS_m_TEMPLATE_PARAMS(...), AND_0_VALUE_PARAMS())`	`Foo<t1, ..., t_m>()`	`FooAction<t1, ..., t_m>`

Given Definition	Expression	Has Type
`ACTION_P(Bar, param)`	`Bar(int_value)`	`BarActionP<int>`
`ACTION_TEMPLATE(Bar, HAS_m_TEMPLATE_PARAMS(...), AND_1_VALUE_PARAMS(p1))`	`Bar<t1, ..., t_m>(int_value)`	`BarActionP<t1, ..., t_m, int>`
`ACTION_P2(Baz, p1, p2)`	`Baz(bool_value, int_value)`	`BazActionP2<bool, int>`
`ACTION_TEMPLATE(Baz, HAS_m_TEMPLATE_PARAMS(...), AND_2_VALUE_PARAMS(p1, p2))`	`Baz<t1, ..., t_m>(bool_value, int_value)`	`BazActionP2<t1, ..., t_m, bool, int>`
...	...	...

Note that we have to pick different suffixes (`Action`, `ActionP`, `ActionP2`, and etc) for actions with different numbers of value parameters, or the action definitions cannot be overloaded on the number of them.

Writing New Monomorphic Actions {#NewMonoActions}

While the `ACTION*` macros are very convenient, sometimes they are inappropriate. For example, despite the tricks shown in the previous recipes, they don't let you directly specify the types of the mock function arguments and the action parameters, which in general leads to unoptimized compiler error messages that can baffle unfamiliar users. They also don't allow overloading actions based on parameter types without jumping through some hoops.

An alternative to the `ACTION*` macros is to implement `::testing::ActionInterface<F>`, where `F` is the type of the mock function in which the action will be used. For example:

```
template <typename F>
class ActionInterface {
 public:
  virtual ~ActionInterface();

  // Performs the action.  Result is the return type of function type
  // F, and ArgumentTuple is the tuple of arguments of F.
  //

  // For example, if F is int(bool, const string&), then Result would
  // be int, and ArgumentTuple would be std::tuple<bool, const string&>.
  virtual Result Perform(const ArgumentTuple& args) = 0;
};
```

```
using ::testing::_;
using ::testing::Action;
using ::testing::ActionInterface;
using ::testing::MakeAction;
```

```
typedef int IncrementMethod(int*);

class IncrementArgumentAction : public ActionInterface<IncrementMethod> {
 public:
  int Perform(const std::tuple<int*>& args) override {
    int* p = std::get<0>(args);  // Grabs the first argument.
    return *p++;
  }
};

Action<IncrementMethod> IncrementArgument() {
  return MakeAction(new IncrementArgumentAction);
}

...
  EXPECT_CALL(foo, Baz(_))
      .WillOnce(IncrementArgument());

  int n = 5;
  foo.Baz(&n);  // Should return 5 and change n to 6.
```

Writing New Polymorphic Actions {#NewPolyActions}

The previous recipe showed you how to define your own action. This is all good, except that you need to know the type of the function in which the action will be used. Sometimes that can be a problem. For example, if you want to use the action in functions with *different* types (e.g. like `Return()` and `SetArgPointee()`).

If an action can be used in several types of mock functions, we say it's *polymorphic*. The `MakePolymorphicAction()` function template makes it easy to define such an action:

```
namespace testing {
template <typename Impl>
PolymorphicAction<Impl> MakePolymorphicAction(const Impl& impl);
}  // namespace testing
```

As an example, let's define an action that returns the second argument in the mock function's argument list. The first step is to define an implementation class:

```
class ReturnSecondArgumentAction {
 public:
  template <typename Result, typename ArgumentTuple>
  Result Perform(const ArgumentTuple& args) const {
    // To get the i-th (0-based) argument, use std::get(args).
    return std::get<1>(args);
  }
};
```

This implementation class does *not* need to inherit from any particular class. What matters is that it must have a `Perform()` method template. This method template takes the mock function's arguments as a tuple in a **single** argument, and returns the result of the action. It can be either `const` or not, but must be invocable with exactly one template argument, which is the result type. In other words, you must be able to call `Perform<R>(args)` where `R` is the mock function's return type and `args` is its arguments in a tuple.

Next, we use `MakePolymorphicAction()` to turn an instance of the implementation class into the polymorphic action we need. It will be convenient to have a wrapper for this:

```
using ::testing::MakePolymorphicAction;
using ::testing::PolymorphicAction;

PolymorphicAction<ReturnSecondArgumentAction> ReturnSecondArgument() {
  return MakePolymorphicAction(ReturnSecondArgumentAction());
}
```

Now, you can use this polymorphic action the same way you use the built-in ones:

```
using ::testing::_;

class MockFoo : public Foo {
 public:
  MOCK_METHOD(int, DoThis, (bool flag, int n), (override));
  MOCK_METHOD(string, DoThat, (int x, const char* str1, const char* str2),
              (override));
};


  ...
  MockFoo foo;
  EXPECT_CALL(foo, DoThis).WillOnce(ReturnSecondArgument());
  EXPECT_CALL(foo, DoThat).WillOnce(ReturnSecondArgument());
  ...
  foo.DoThis(true, 5);  // Will return 5.
  foo.DoThat(1, "Hi", "Bye");  // Will return "Hi".
```

Teaching gMock How to Print Your Values

When an uninteresting or unexpected call occurs, gMock prints the argument values and the stack trace to help you debug. Assertion macros like `EXPECT_THAT` and `EXPECT_EQ` also print the values in question when the assertion fails. gMock and googletest do this using googletest's user-extensible value printer.

This printer knows how to print built-in C++ types, native arrays, STL containers, and any type that supports the `<<` operator. For other types, it prints the raw bytes in the value and hopes that you the user can figure it out. The GoogleTest advanced guide
explains how to extend the printer to do a better job at printing your particular type than to dump the bytes.

Useful Mocks Created Using gMock

Mock std::function {#MockFunction}

`std::function` is a general function type introduced in C++11. It is a preferred way of passing callbacks to new interfaces. Functions are copiable, and are not usually passed around by pointer, which makes them tricky to mock. But fear not - `MockFunction` can help you with that.

`MockFunction<R(T1, ..., Tn)>` has a mock method `Call()` with the signature:

```
  R Call(T1, ..., Tn);
```

It also has a `AsStdFunction()` method, which creates a `std::function` proxy forwarding to Call:

```
  std::function<R(T1, ..., Tn)> AsStdFunction();
```

To use `MockFunction`, first create `MockFunction` object and set up expectations on its `Call` method. Then pass proxy obtained from `AsStdFunction()` to the code you are testing. For example:

```
TEST(FooTest, RunsCallbackWithBarArgument) {
  // 1. Create a mock object.
  MockFunction<int(string)> mock_function;

  // 2. Set expectations on Call() method.
  EXPECT_CALL(mock_function, Call("bar")).WillOnce(Return(1));

  // 3. Exercise code that uses std::function.
  Foo(mock_function.AsStdFunction());
  // Foo's signature can be either of:
  // void Foo(const std::function<int(string)>& fun);
  // void Foo(std::function<int(string)> fun);

  // 4. All expectations will be verified when mock_function
  //     goes out of scope and is destroyed.
}
```

Remember that function objects created with `AsStdFunction()` are just forwarders. If you create multiple of them, they will share the same set of expectations.

Although `std::function` supports unlimited number of arguments, `MockFunction` implementation is limited to ten. If you ever hit that limit... well, your callback has bigger problems than being mockable. :-)