google / **googletest**

Branch: **master** **googletest** / googlemock / docs / **FrequentlyAskedQuestions.md** Find file Copy path

**Paul Rosset** fixed link in googlemock documentation a470862 on 8 Dec 2015

**1** contributor

```
629 lines (516 sloc)   22.8 KB
```

Please send your questions to the googlemock discussion group. If you need help with compiler errors, make sure you have tried Google Mock Doctor first.

## When I call a method on my mock object, the method for the real object is invoked instead. What's the problem?

In order for a method to be mocked, it must be *virtual*, unless you use the high-perf dependency injection technique.

## I wrote some matchers. After I upgraded to a new version of Google Mock, they no longer compile. What's going on?

After version 1.4.0 of Google Mock was released, we had an idea on how to make it easier to write matchers that can generate informative messages efficiently. We experimented with this idea and liked what we saw. Therefore we decided to implement it.

Unfortunately, this means that if you have defined your own matchers by implementing `MatcherInterface` or using `MakePolymorphicMatcher()`, your definitions will no longer compile. Matchers defined using the `MATCHER*` family of macros are not affected.

Sorry for the hassle if your matchers are affected. We believe it's in everyone's long-term interest to make this change sooner than later. Fortunately, it's usually not hard to migrate an existing matcher to the new API. Here's what you need to do:

If you wrote your matcher like this:

```
// Old matcher definition that doesn't work with the latest
// Google Mock.
using ::testing::MatcherInterface;
...
class MyWonderfulMatcher : public MatcherInterface<MyType> {
 public:
  ...
  virtual bool Matches(MyType value) const {
    // Returns true if value matches.
    return value.GetFoo() > 5;
  }
  ...
};
```

you'll need to change it to:

```
// New matcher definition that works with the latest Google Mock.
using ::testing::MatcherInterface;
using ::testing::MatchResultListener;
...
```

```
class MyWonderfulMatcher : public MatcherInterface<MyType> {
 public:
  ...
  virtual bool MatchAndExplain(MyType value,
                               MatchResultListener* listener) const {
    // Returns true if value matches.
    return value.GetFoo() > 5;
  }
  ...
};
```

(i.e. rename `Matches()` to `MatchAndExplain()` and give it a second argument of type `MatchResultListener*` .)

If you were also using `ExplainMatchResultTo()` to improve the matcher message:

```
// Old matcher definition that doesn't work with the lastest
// Google Mock.
using ::testing::MatcherInterface;
...
class MyWonderfulMatcher : public MatcherInterface<MyType> {
 public:
  ...
  virtual bool Matches(MyType value) const {
    // Returns true if value matches.
    return value.GetFoo() > 5;
  }

  virtual void ExplainMatchResultTo(MyType value,
                                    ::std::ostream* os) const {
    // Prints some helpful information to os to help
    // a user understand why value matches (or doesn't match).
    *os << "the Foo property is " << value.GetFoo();
  }
  ...
};
```

you should move the logic of `ExplainMatchResultTo()` into `MatchAndExplain()` , using the `MatchResultListener` argument where the `::std::ostream` was used:

```
// New matcher definition that works with the latest Google Mock.
using ::testing::MatcherInterface;
using ::testing::MatchResultListener;
...
class MyWonderfulMatcher : public MatcherInterface<MyType> {
 public:
  ...
  virtual bool MatchAndExplain(MyType value,
                               MatchResultListener* listener) const {
    // Returns true if value matches.
    *listener << "the Foo property is " << value.GetFoo();
    return value.GetFoo() > 5;
  }
  ...
};
```

If your matcher is defined using `MakePolymorphicMatcher()` :

```
// Old matcher definition that doesn't work with the latest
// Google Mock.
using ::testing::MakePolymorphicMatcher;
...
class MyGreatMatcher {
 public:
  ...
  bool Matches(MyType value) const {
    // Returns true if value matches.
    return value.GetBar() < 42;
  }
  ...
};
... MakePolymorphicMatcher(MyGreatMatcher()) ...
```

you should rename the `Matches()` method to `MatchAndExplain()` and add a `MatchResultListener*` argument (the same as what you need to do for matchers defined by implementing `MatcherInterface` ):

```
// New matcher definition that works with the latest Google Mock.
using ::testing::MakePolymorphicMatcher;
using ::testing::MatchResultListener;
...
class MyGreatMatcher {
 public:
  ...
  bool MatchAndExplain(MyType value,
                       MatchResultListener* listener) const {
    // Returns true if value matches.
    return value.GetBar() < 42;
  }
  ...
};
... MakePolymorphicMatcher(MyGreatMatcher()) ...
```

If your polymorphic matcher uses `ExplainMatchResultTo()` for better failure messages:

```
// Old matcher definition that doesn't work with the latest
// Google Mock.
using ::testing::MakePolymorphicMatcher;
...
class MyGreatMatcher {
 public:
  ...
  bool Matches(MyType value) const {
    // Returns true if value matches.
    return value.GetBar() < 42;
  }
  ...
};
void ExplainMatchResultTo(const MyGreatMatcher& matcher,
                          MyType value,
                          ::std::ostream* os) {
  // Prints some helpful information to os to help
  // a user understand why value matches (or doesn't match).
  *os << "the Bar property is " << value.GetBar();
}
... MakePolymorphicMatcher(MyGreatMatcher()) ...
```

you'll need to move the logic inside `ExplainMatchResultTo()` to `MatchAndExplain()` :

```
// New matcher definition that works with the latest Google Mock.
using ::testing::MakePolymorphicMatcher;
using ::testing::MatchResultListener;
...
class MyGreatMatcher {
 public:
  ...
```

```
    bool MatchAndExplain(MyType value,
                         MatchResultListener* listener) const {
      // Returns true if value matches.
      *listener << "the Bar property is " << value.GetBar();
      return value.GetBar() < 42;
    }
    ...
  };
  ... MakePolymorphicMatcher(MyGreatMatcher()) ...
```

For more information, you can read these two recipes from the cookbook. As always, you are welcome to post questions on `googlemock@googlegroups.com` if you need any help.

## When using Google Mock, do I have to use Google Test as the testing framework? I have my favorite testing framework and don't want to switch.

Google Mock works out of the box with Google Test. However, it's easy to configure it to work with any testing framework of your choice. Here is how.

## How am I supposed to make sense of these horrible template errors?

If you are confused by the compiler errors gcc threw at you, try consulting the *Google Mock Doctor* tool first. What it does is to scan stdin for gcc error messages, and spit out diagnoses on the problems (we call them diseases) your code has.

To "install", run command:

```
alias gmd='<path to googlemock>/scripts/gmock_doctor.py'
```

To use it, do:

```
<your-favorite-build-command> <your-test> 2>&1 | gmd
```

For example:

```
make my_test 2>&1 | gmd
```

Or you can run `gmd` and copy-n-paste gcc's error messages to it.

## Can I mock a variadic function?

You cannot mock a variadic function (i.e. a function taking ellipsis ( `...` ) arguments) directly in Google Mock.

The problem is that in general, there is *no way* for a mock object to know how many arguments are passed to the variadic method, and what the arguments' types are. Only the *author of the base class* knows the protocol, and we cannot look into his head.

Therefore, to mock such a function, the *user* must teach the mock object how to figure out the number of arguments and their types. One way to do it is to provide overloaded versions of the function.

Ellipsis arguments are inherited from C and not really a C++ feature. They are unsafe to use and don't work with arguments that have constructors or destructors. Therefore we recommend to avoid them in C++ as much as possible.

## MSVC gives me warning C4301 or C4373 when I define a mock method with a const parameter. Why?

If you compile this using Microsoft Visual C++ 2005 SP1:

```
class Foo {
  ...
  virtual void Bar(const int i) = 0;
};

class MockFoo : public Foo {
  ...
  MOCK_METHOD1(Bar, void(const int i));
};
```

You may get the following warning:

```
warning C4301: 'MockFoo::Bar': overriding virtual function only differs from 'Foo::Bar' by
const/volatile qualifier
```

This is a MSVC bug. The same code compiles fine with gcc ,for example. If you use Visual C++ 2008 SP1, you would get the warning:

```
warning C4373: 'MockFoo::Bar': virtual function overrides 'Foo::Bar', previous versions of the compiler
did not override when parameters only differed by const/volatile qualifiers
```

In C++, if you *declare* a function with a `const` parameter, the `const` modifier is *ignored*. Therefore, the `Foo` base class above is equivalent to:

```
class Foo {
  ...
  virtual void Bar(int i) = 0;  // int or const int?  Makes no difference.
};
```

In fact, you can *declare* Bar() with an `int` parameter, and *define* it with a `const int` parameter. The compiler will still match them up.

Since making a parameter `const` is meaningless in the method *declaration*, we recommend to remove it in both `Foo` and `MockFoo`. That should workaround the VC bug.

Note that we are talking about the *top-level* `const` modifier here. If the function parameter is passed by pointer or reference, declaring the *pointee* or *referee* as `const` is still meaningful. For example, the following two declarations are *not* equivalent:

```
void Bar(int* p);        // Neither p nor *p is const.
void Bar(const int* p);  // p is not const, but *p is.
```

## I have a huge mock class, and Microsoft Visual C++ runs out of memory when compiling it. What can I do?

We've noticed that when the `/clr` compiler flag is used, Visual C++ uses 5~6 times as much memory when compiling a mock class. We suggest to avoid `/clr` when compiling native C++ mocks.

## I can't figure out why Google Mock thinks my expectations are not satisfied. What should I do?

You might want to run your test with `--gmock_verbose=info`. This flag lets Google Mock print a trace of every mock function call it receives. By studying the trace, you'll gain insights on why the expectations you set are not met.

## How can I assert that a function is NEVER called?

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

## I have a failed test where Google Mock tells me TWICE that a particular expectation is not satisfied. Isn't this redundant?

When Google Mock detects a failure, it prints relevant information (the mock function arguments, the state of relevant expectations, and etc) to help the user debug. If another failure is detected, Google Mock will do the same, including printing the state of relevant expectations.

Sometimes an expectation's state didn't change between two failures, and you'll see the same description of the state twice. They are however *not* redundant, as they refer to *different points in time*. The fact they are the same *is* interesting information.

## I get a heap check failure when using a mock object, but using a real object is fine. What can be wrong?

Does the class (hopefully a pure interface) you are mocking have a virtual destructor?

Whenever you derive from a base class, make sure its destructor is virtual. Otherwise Bad Things will happen. Consider the following code:

```
class Base {
 public:
  // Not virtual, but should be.
  ~Base() { ... }
  ...
};

class Derived : public Base {
 public:
  ...
 private:
  std::string value_;
};

...
  Base* p = new Derived;
  ...
  delete p;  // Surprise! ~Base() will be called, but ~Derived() will not
             // - value_ is leaked.
```

By changing `~Base()` to virtual, `~Derived()` will be correctly called when `delete p` is executed, and the heap checker will be happy.

## The "newer expectations override older ones" rule makes writing expectations awkward. Why does Google Mock do that?

When people complain about this, often they are referring to code like:

```
// foo.Bar() should be called twice, return 1 the first time, and return
// 2 the second time.  However, I have to write the expectations in the
// reverse order.  This sucks big time!!!
EXPECT_CALL(foo, Bar())
    .WillOnce(Return(2))
    .RetiresOnSaturation();
EXPECT_CALL(foo, Bar())
    .WillOnce(Return(1))
    .RetiresOnSaturation();
```

The problem is that they didn't pick the **best** way to express the test's intent.

By default, expectations don't have to be matched in *any* particular order. If you want them to match in a certain order, you need to be explicit. This is Google Mock's (and jMock's) fundamental philosophy: it's easy to accidentally over-specify your tests, and we want to make it harder to do so.

There are two better ways to write the test spec. You could either put the expectations in sequence:

```
// foo.Bar() should be called twice, return 1 the first time, and return
// 2 the second time.  Using a sequence, we can write the expectations
// in their natural order.
{
  InSequence s;
  EXPECT_CALL(foo, Bar())
      .WillOnce(Return(1))
      .RetiresOnSaturation();
  EXPECT_CALL(foo, Bar())
      .WillOnce(Return(2))
      .RetiresOnSaturation();
}
```

or you can put the sequence of actions in the same expectation:

```
// foo.Bar() should be called twice, return 1 the first time, and return
// 2 the second time.
EXPECT_CALL(foo, Bar())
    .WillOnce(Return(1))
    .WillOnce(Return(2))
    .RetiresOnSaturation();
```

Back to the original questions: why does Google Mock search the expectations (and `ON_CALL` s) from back to front? Because this allows a user to set up a mock's behavior for the common case early (e.g. in the mock's constructor or the test fixture's set-up phase) and customize it with more specific rules later. If Google Mock searches from front to back, this very useful pattern won't be possible.

## Google Mock prints a warning when a function without EXPECT_CALL is called, even if I have set its behavior using ON_CALL. Would it be reasonable not to show the warning in this case?

When choosing between being neat and being safe, we lean toward the latter. So the answer is that we think it's better to show the warning.

Often people write `ON_CALL` s in the mock object's constructor or `SetUp()` , as the default behavior rarely changes from test to test. Then in the test body they set the expectations, which are often different for each test. Having an `ON_CALL` in the set-up part of a test doesn't mean that the calls are expected. If there's no `EXPECT_CALL` and the method is called, it's possibly an error. If we quietly let the call go through without notifying the user, bugs may creep in unnoticed.

If, however, you are sure that the calls are OK, you can write

```
EXPECT_CALL(foo, Bar(_))
    .WillRepeatedly(...);
```

instead of

```
ON_CALL(foo, Bar(_))
    .WillByDefault(...);
```

This tells Google Mock that you do expect the calls and no warning should be printed.

Also, you can control the verbosity using the `--gmock_verbose` flag. If you find the output too noisy when debugging, just choose a less verbose level.

## How can I delete the mock function's argument in an action?

If you find yourself needing to perform some action that's not supported by Google Mock directly, remember that you can define your own actions using MakeAction() or MakePolymorphicAction(), or you can write a stub function and invoke it using Invoke().

## MOCK_METHODn()'s second argument looks funny. Why don't you use the MOCK_METHODn(Method, return_type, arg_1, ..., arg_n) syntax?

What?! I think it's beautiful. :-)

While which syntax looks more natural is a subjective matter to some extent, Google Mock's syntax was chosen for several practical advantages it has.

Try to mock a function that takes a map as an argument:

```
virtual int GetSize(const map<int, std::string>& m);
```

Using the proposed syntax, it would be:

```
MOCK_METHOD1(GetSize, int, const map<int, std::string>& m);
```

Guess what? You'll get a compiler error as the compiler thinks that `const map<int, std::string>& m` are **two**, not one, arguments. To work around this you can use `typedef` to give the map type a name, but that gets in the way of your work. Google Mock's syntax avoids this problem as the function's argument types are protected inside a pair of parentheses:

```
// This compiles fine.
MOCK_METHOD1(GetSize, int(const map<int, std::string>& m));
```

You still need a `typedef` if the return type contains an unprotected comma, but that's much rarer.

Other advantages include:

1. `MOCK_METHOD1(Foo, int, bool)` can leave a reader wonder whether the method returns `int` or `bool` , while there won't be such confusion using Google Mock's syntax.
2. The way Google Mock describes a function type is nothing new, although many people may not be familiar with it. The same syntax was used in C, and the `function` library in `tr1` uses this syntax extensively. Since `tr1` will become a part of the new version of STL, we feel very comfortable to be consistent with it.
3. The function type syntax is also used in other parts of Google Mock's API (e.g. the action interface) in order to make the implementation tractable. A user needs to learn it anyway in order to utilize Google Mock's more advanced features. We'd as well stick to the same syntax in `MOCK_METHOD*` !

## My code calls a static/global function. Can I mock it?

You can, but you need to make some changes.

In general, if you find yourself needing to mock a static function, it's a sign that your modules are too tightly coupled (and less flexible, less reusable, less testable, etc). You are probably better off defining a small interface and call the function through that interface, which then can be easily mocked. It's a bit of work initially, but usually pays for itself quickly.

This Google Testing Blog post says it excellently. Check it out.

## My mock object needs to do complex stuff. It's a lot of pain to specify the actions. Google Mock sucks!

I know it's not a question, but you get an answer for free any way. :-)

With Google Mock, you can create mocks in C++ easily. And people might be tempted to use them everywhere. Sometimes they work great, and sometimes you may find them, well, a pain to use. So, what's wrong in the latter case?

When you write a test without using mocks, you exercise the code and assert that it returns the correct value or that the system is in an expected state. This is sometimes called "state-based testing".

Mocks are great for what some call "interaction-based" testing: instead of checking the system state at the very end, mock objects verify that they are invoked the right way and report an error as soon as it arises, giving you a handle on the precise context in which the error was triggered. This is often more effective and economical to do than state-based testing.

If you are doing state-based testing and using a test double just to simulate the real object, you are probably better off using a fake. Using a mock in this case causes pain, as it's not a strong point for mocks to perform complex actions. If you experience this and think that mocks suck, you are just not using the right tool for your problem. Or, you might be trying to solve the wrong problem. :-)

## I got a warning "Uninteresting function call encountered - default action taken.." Should I panic?

By all means, NO! It's just an FYI.

What it means is that you have a mock function, you haven't set any expectations on it (by Google Mock's rule this means that you are not interested in calls to this function and therefore it can be called any number of times), and it is called. That's OK - you didn't say it's not OK to call the function!

What if you actually meant to disallow this function to be called, but forgot to write `EXPECT_CALL(foo, Bar()).Times(0)` ? While one can argue that it's the user's fault, Google Mock tries to be nice and prints you a note.

So, when you see the message and believe that there shouldn't be any uninteresting calls, you should investigate what's going on. To make your life easier, Google Mock prints the function name and arguments when an uninteresting call is encountered.

## I want to define a custom action. Should I use Invoke() or implement the action interface?

Either way is fine - you want to choose the one that's more convenient for your circumstance.

Usually, if your action is for a particular function type, defining it using `Invoke()` should be easier; if your action can be used in functions of different types (e.g. if you are defining `Return(value)` ), `MakePolymorphicAction()` is easiest. Sometimes you want precise control on what types of functions the action can be used in, and implementing `ActionInterface` is the way to go here. See the implementation of `Return()` in `include/gmock/gmock-actions.h` for an example.

## I'm using the set-argument-pointee action, and the compiler complains about "conflicting return type specified". What does it mean?

You got this error as Google Mock has no idea what value it should return when the mock method is called. `SetArgPointee()` says what the side effect is, but doesn't say what the return value should be. You need `DoAll()` to chain a `SetArgPointee()` with a `Return()`.

See this recipe for more details and an example.

## My question is not in your FAQ!

If you cannot find the answer to your question in this FAQ, there are some other resources you can use:

1. read other documentation,
2. search the mailing list archive,
3. ask it on googlemock@googlegroups.com and someone will answer it (to prevent spam, we require you to join the discussion group before you can post.).

Please note that creating an issue in the issue tracker is *not* a good way to get your answer, as it is monitored infrequently by a very small number of people.

When asking a question, it's helpful to provide as much of the following information as possible (people cannot help you if there's not enough information in your question):

- the version (or the revision number if you check out from SVN directly) of Google Mock you use (Google Mock is under active development, so it's possible that your problem has been solved in a later version),
- your operating system,
- the name and version of your compiler,
- the complete command line flags you give to your compiler,
- the complete compiler error messages (if the question is about compilation),
- the *actual* code (ideally, a minimal but complete program) that has the problem you encounter.