# Legacy gMock FAQ

## 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.

## Can I mock a variadic function?

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

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 or her 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_METHOD(void, Bar, (const int i), (override));
};
```

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

You might want to run your test with `--gmock_verbose=info`. This flag lets gMock 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.

If you see the message "The mock function has no default action set, and its return type has no default value set.", then try adding a default action. Due to a known issue, unexpected calls on mocks without default actions don't print out a detailed comparison between the actual arguments and the expected arguments.

## My program crashed and `ScopedMockLog` spit out tons of messages. Is it a gMock bug?

gMock and `ScopedMockLog` are likely doing the right thing here.

When a test crashes, the failure signal handler will try to log a lot of information (the stack trace, and the address map, for example). The messages are compounded if you have many threads with depth stacks. When `ScopedMockLog` intercepts these messages and finds that they don't match any expectations, it prints an error for each of them.

You can learn to ignore the errors, or you can rewrite your expectations to make your test more robust, for example, by adding something like:

```
using ::testing::AnyNumber;
using ::testing::Not;
...
  // Ignores any log not done by us.
  EXPECT_CALL(log, Log(_, Not(EndsWith("/my_file.cc")), _))
      .Times(AnyNumber());
```

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

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

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

When gMock 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, gMock 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 heapcheck 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 gMock do that?

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

```
using ::testing::Return;
...
  // 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 gMock'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:

```
using ::testing::Return;
...
  // 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:

```
using ::testing::Return;
...
  // 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 gMock 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 gMock searches from front to back, this very useful pattern won't be possible.

## gMock 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

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

instead of

```
using ::testing::_;
...
  ON_CALL(foo, Bar(_))
      .WillByDefault(...);
```

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

Also, you can control the verbosity by specifying `--gmock_verbose=error`. Other values are `info` and `warning`. 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 your mock function takes a pointer argument and you want to delete that argument, you can use testing::DeleteArg() to delete the N'th (zero-indexed) argument:

```
using ::testing::_;
  ...
  MOCK_METHOD(void, Bar, (X* x, const Y& y));
  ...
  EXPECT_CALL(mock_foo_, Bar(_, _))
      .WillOnce(testing::DeleteArg<0>()));
```

## How can I perform an arbitrary action on a mock function's argument?

If you find yourself needing to perform some action that's not supported by gMock 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()`.

```
using ::testing::_;
using ::testing::Invoke;
  ...
  MOCK_METHOD(void, Bar, (X* p));
  ...
  EXPECT_CALL(mock_foo_, Bar(_))
      .WillOnce(Invoke(MyAction(...)));
```

## 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. gMock sucks!

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

With gMock, 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 gMock'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, gMock 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, gMock dumps the stack trace when an uninteresting call is encountered. From that you can figure out which mock function it is, and how it is called.

## I want to define a custom action. Should I use Invoke() or implement the ActionInterface 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 `gmock-actions.h` for an example.

## I use SetArgPointee() in WillOnce(), but gcc complains about "conflicting return type specified". What does it mean?

You got this error as gMock 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()` that provides a value appropriate to the API being mocked.

See this recipe for more details and an example.

## 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.