Tutorial 3 0

0. Changes from 2.0

Breaking changes:

- The ampersand before a function name is no longer required & allowed. Since you'd have to type an ampersand anyway, we decided to put that into the mocking framework for you. The unreadable and indecipherable error associated with this (reported by Sean DeNigris) are now also impossible.

Bugfixes:

- Returning a reference or a const-reference now works. Returning a (non-const) reference to a temporary is not allowed.

Additions:

- It's now possible to inform the mocking framework that a specific function should never be called. While this doesn't fundamentally change the behaviour, it does allow much more thorough error reporting.
- Error reports now include as much information as the mocking framework can provide. This includes all function registrations and all related information, including printing the arguments in a readable format (even those that cannot be printed).
- Added comments on each warning disabled for Visual Studio why it is being disabled. Comments are basically my (personal) reaction to what the warning tries to say.

1. Basic usage

Say, we have a class called Foo.

```
class Foo {
private:
    IBar *bar;
public:
    Foo(IBar *bar);
    int a();
};
```

We want to test the a() function. It uses an object with the interface IBar.

```
class IBar {
public:
    virtual ~IBar() {}
    virtual void b() = 0;
    virtual int c(std::string) = 0;
};
```

It is some kind of interface crucial to the function of the a() function in Foo, but we really would rather not use the real Bar objects because:

- It is large
- It is slow
- It is not tested
- It uses external stuff (files, databases, network)
- It does not exist yet
- It costs money to use (thanks to Daniel Meyer and his blog for the idea)
- It is not relevant to Foo's internal logic and to its tests

So, we want to make a test for Foo without using Bar objects. We can use anything instead of Bar that satisfies the IBar interface, so we create a mock repository and tell it to give us a IBar implementing object of otherwise unspecified contents:

```
void TestAFunctionInFoo() {
    MockRepository mocks;
    IBar *barMock = mocks.InterfaceMock<IBar>();
```

Now we have an IBar and we can create the Foo to use it:

```
    Foo *newFoo = new Foo(barMock);
```

We know that Foo doesn't use its IBar during it's construction. Now, we register the expectations with the mock repository, so it knows what to expect. If we call something we didn't mention before, we get a regular exception informing us of that. If we on the other hand call something we did mention before, it can be checked for validity. We expect a call on b().

```
    mocks.ExpectCall(barMock, IBar::b);
```

We also expect a call on c(), with "hello" as argument, upon which it should return 42. Let's inform the mock repository of this information:

```
    mocks.ExpectCall(barMock, IBar::c).With("hello").Return(42);
```

That reads a lot like the sentences you just read describing what we were going to do - actually, you can read this out loud as regular text. The mock repository arms all expectations immediately, so we can now execute the test itself and let it call the mock functions.

```
    newFoo->a();
```

We close up the test case, letting the mock repository clean up all the mocks and related stuff:

```
    delete newFoo;
}
```

This last line (the accolade) tells the mock repository to verify its expectations. If an error already occurred, it will not throw a second exception.

Notice that in the process of making this test, we don't know Bar at all. Bar might not exist, might be unusable or might be developed on the other side of the planet. We have been able to test our code, without knowing Bar or anything about Bar except that it implements IBar.

This completes the first bit of the tutorial. To make this compile, include and include "hippomocks.h", and it will compile and work on at least GCC and Visual Studio.

2. More complex stuff

We can mock more than just functions with return values, that have to be called, and that always execute in the same order. The basic call:

```
mocks.ExpectCall(barMock, IBar::c).With("hello").Return(42);
```

We can tell the mocking framework a function does not return a value but throws an exception:

```
mocks.ExpectCall(barMock, IBar::c).With("hello").Throw(std::exception());
```

We can tell it that a function may be called, but doesn't have to be called:

```
mocks.OnCall(barMock, IBar::c).With("hello").Return(42);
```

Maybe we don't care about the arguments to a function

```
mocks.ExpectCall(barMock, IBar::c).Return(42);
```

Maybe we want to substitute the entire function body?

```
int someOtherFunction(std::string);
mocks.ExpectCall(barMock, IBar::c).With("hello").Do(someOtherFunction);
```

What if we expect multiple, different calls?

```
mocks.ExpectCall(barMock, IBar::c).With("hello").Return(42);
mocks.ExpectCall(barMock, IBar::c).With("world").Return(42);
```

And if we don't care about the order?

```
mocks.autoExpect = false;
mocks.ExpectCall(barMock, IBar::c).With("hello").Return(42);
mocks.ExpectCall(barMock, IBar::c).With("world").Return(42);
```

And what if we do care about the order, but only between certain function calls? (yes, we're going to make this hard)

```
mocks.autoExpect = false;
Call &callOne = mocks.ExpectCall(barMock, IBar::c).With("hello").Return(1);
Call &callTwo = mocks.ExpectCall(barMock, IBar::c).With("world").Return(2);
mocks.ExpectCall(barMock, IBar::c).After(callOne).With("1").Return(3);
mocks.ExpectCall(barMock, IBar::c).After(callTwo).With("2").Return(4);
```

When we have a class that has members, how do we make sure it's constructed?

```
Bar *barMock = mocks.ClassMock<Bar>();
```

And when Bar has both pure virtual functions and members?

```
// there's a limit in the language that disallows us from making a mock of this, as we can't call the constructor and we can't skip calling the constructor
// The only solution at the moment is to derive a class from it that implements the pure virtuals and then to make a mock out of that. The implementati
on
// will not be used at all, ever, except for calling the constructor.

class BarImpl : public Bar {
public:
    virtual void b() {}
};
Bar *barMock = mocks.ClassMock<BarImpl>();
```

3. Advanced stuff (or, new in this version)

What if we want to change the way you compare types? Well... overload operator== for the type. If that's not possible (example: compare strings with strcmp) use this:

```
// put this after including hippomocks.h, or modify hippomocks.h
template <>
struct comparer<const char *>
{
  static inline bool compare(const char *a, const char *b)
  {
    return strcmp(a, b) == 0;
  }
};
```

What if we want to change the way you print types? Well... overload operator<< for ostream& and your type and make it output the right way. If that's not possible (example: char type as int):

```
// put this after including hippomocks.h, or modify hippomocks.h
template <>
struct printArg<unsigned char>
{
  static inline void print(std::ostream &os, unsigned char arg, bool withComma)
  {
    if (withComma) os << ",";
    os << (char)arg;
  }
};
```

What if we want to indicate a function is not called, for example after using another function?

```
// For NeverCalls all the same restrictions and combinations hold that you get with ExpectCall. They're tried after ExpectCalls and before OnCalls.
Call &callOne = mocks.ExpectCall(barMock, IBar::c).With("hello").Return(1);
mocks.NeverCall(barMock, IBar::c).With("1").After(callOne);
```

3.1. New in release 3.1

What if I want to reduce memory consumption?

```
Change the following defines to a smaller number before including hippomocks.h (either before including it or on the command-line):
EXCEPTION_BUFFER_SIZE : The maximum size of the message in an exception.
    This defaults to 64k, which is smallish on Windows-based platforms and just about always enough.
    Values between 1 and 4 kilobytes are probably sufficient for embedded platforms.
VIRT_FUNC_LIMIT : The maximum amount of virtual functions in a class. This defaults to 1024.
    Most designs will remain below 16 functions per class or so, with some exceptions going up to 50-100 per class.
    If you know your own limit you can reduce this by a lot, saving memory for each mock object created.
Turn on optimizations. Lots of code in Hippo Mocks is suited for inlining to save space.
```

What if I have two functions with the same name, but a different signature?

```
Add Overload to the type of registration and cast your function name to the right type. For example, adding:

    virtual int c(int) = 0;

to IBar breaks the mocking of c, as it doesn't know which to take. By changing the ExpectCall to:

    mocks.ExpectCallOverload(barMock, (int(IBar::*)(int))&IBar::c).With("hello").Return(42);

the compiler is satisfied again. Note that the type for the member needs to be entirely specified and that the ampersand in front of the name is required
.
```

If you can think of a use case that is not clear or easy with these constructs, please tell us!