LegacyMock User's Guide

If your C++ code wasn't testable before, it is now!

The user's guide is intended for other developers who want to use LegacyMock to test their own C++ applications. Therefore, it will not go into details about how LegacyMock works. We will provide a developer's guide at some later date about how to contribute to LegacyMock and thus also go more deeply into inner workings and concepts there.

First, we will start with describing some scenarios which commonly exist all over the place in legacy code and how LegacyMock will help you test such code anyway without breaking a sweat. After discussing a bunch of more advanced features, we will describe how you can integrate it into your own unit tests and what to watch out for concerning the continuous updates to LegacyMock that will arrive.

Contents

1.1 Introduction

LegacyMock was developed out of the need for testing legacy code, which by definition lacks proper UnitTest coverage. The problem is, how to add tests? Without good test coverage any change to the existing codebase is a potential bug. Without code change, there is often no way to add tests, since legacy code is written in a way that is closed for modification and closed for extension. To break this cycle of doom there are three options:

1) Rewrite the legacy code module by module from scratch. Don't do this at home!
2) Start carefully modifying the codebase and make it testable. Even this is likely to get very bumpy and close to impossible. Retrofitting extensibility through interfaces and composition into existing hostile legacy code often requires massive refactoring to even get a small increase in testability.
3) Don't modify the codebase, just add the tests. This is what LegacyMock allows you to do.

Choosing the third option has a few advantages:

1) You simply *can't* introduce bugs by adding tests, which is a great relief (fear of change).
2) People who are not familiar with your software can write the tests and get on-boarded by simply "doing" instead of worrying, as they can't possibly break anything.
3) Once you have the test coverage, LegacyMock's interface based approach allows you to simply swap out legacy classes with interfaces (the massive refactoring mentioned earlier) without even changing the test-cases (in the ideal case).
4) You get interfaces and Google Mocks for free (don't need to write them yourself, "Don't repeat yourself!").
5) Very low entry barrier. You can just start writing some tests without worrying about anything.
6) LegacyMock is simply Google Mock on steroids and anyone who is familiar with GMock will feel right at home.

So let's get ready to do some testing…

1.2 Compatibility

LegacyMock did once support all the platforms and compilers listed here. But since we wanted to get a first prototype out more quickly we started to focus on Windows, but all the listed platforms will eventually be supported again.

Which operating systems does LegacyMock support?

- Microsoft Windows
- Mac OS & Linux (not implemented)

Which Compilers does it support?

- Visual Studio 2005 or later

- GCC 4.2 or later (not implemented)
- XCode 4 or later (not implemented)

Further it is known to work with Boost 1.32 or higher and GTest/GMock 1.5 or higher. Note that call tracing is only supported with Boost 1.41 or higher.

1.3 A Simple Example of Untestable Code

Consider the following legacy code and its futile attempt of testing. The goal is to test the implementation of `myLegacyFunc` without changing its source code. `MyLegacyStruct` in contrast is a data-structure we already have tested elsewhere. This means, it should be mocked. Unfortunately, that turns out to be difficult:

```
// suppose you can't touch this code…
class MyLegacyStruct {
public:
    void print(std::string message) {
        std::cout << message << std::endl;
    }
};

// suppose you can't touch this code…
void myLegacyFunc(MyLegacyStruct& value) {
    value.print("Hello World!");
}

TEST(Samples, IWantToTestThat) {
    // doh?!
    MyLegacyStruct instance;

    myLegacyFunc(instance);
}
```

How does LegacyMock help?

```
TEST(Samples, IWantToTestThat) {
    G::MockFixture fixture;
    G::Mock<MyLegacyStruct> mock;

    EXPECT_CALL(mock, print("Hello World!"))
        .Times(1);

    myLegacyFunc(mock);

    // don't have a cow!
}
```

The `G::MockFixture` sets the scope for the test that follows and can be added to the Google Mock fixture, if you use one. It is essential to have all mock calls inside an active fixture (and a fresh fixture for every test), otherwise it will refuse to work or produce wrong results.

`G::Mock<MyLegacyStruct>` does two important things. It is basically a Google Mock wrapper around our `MyLegacyStruct` class and, as you can see, seamlessly allows for interaction with the normal Google Mock routines. Further, it allocates an actual instance of the object it wraps around (and yes it forwards the constructor arguments you give, to the actual constructor), which allows you to pass it into legacy code that knows nothing about GMock or our mock object. This magic is internally realized through compile time code generation and runtime machine code patching, both done by LegacyMock in the background.

If now the `print` statement is executed, LegacyMock will route that call into Google Mock, which will see that we have defined a nice expectation that this call satisfies.

Now this wasn't that much magic, because somehow you could cheat & hack your way to this without using a special framework for it in some cases. So let's look at a more complicated case.

1.4 A Not So Simple Example of Untestable Code

The main difference here is that the function you want to mock does not allow you to pass the mock into it, but rather allocates a fixed instance inside of itself.

```
// suppose you can't touch this code…
class MyLegacyStruct {
public:
    void print(std::string message) {
        std::cout << message << std::endl;
    }
};

void myLegacyFunc() {
    MyLegacyStruct value;
    value.print("Hello World!");
}

// … and this was the only thing you can modify/provide
TEST(Samples, IWantToTestThat) {
    myLegacyFunc();
}
```

This will be impossible to handle without changing the code in some way. How does LegacyMock help?

```
TEST(Samples, IWantToTestThat) {
    G::MockFixture fixture;
    G::LazyMock<MyLegacyStruct> lazy;

    EXPECT_CALL(lazy, print("Hello World!"))
        .Times(1);

    myLegacyFunc();
```

```
}
```

The one major change we made is using `G::LazyMock` instead of `G::Mock`. A lazy mock, as its name suggests, will *not* create an instance of the wrapped object. Every object of the wrapped type that is created later (copy constructor, new, make_shared, etc.), will bind to a queued lazy mock (FIFO). If there are no more lazy mocks, the testcase will fail.

Lazy mocks are very powerful, but also very difficult. As soon as you deal with *modern* C++ (because it encourages passing by value and rvalue references among other things) lazy binding can be very annoying because each of those actions will require a new mock object, which makes your testcase highly dependent on the code structure and thus actually harmful. Currently, the only supported alternative is a `G::DefaultMock`, which receives all invocations if there are no other mock types available. Not always is this enough to write a proper test-case, which is why we are investigating other mock types, which do not yet exist, tracking such *fundamental* operations (think of `G::TrackedMock` & `G::TrackedLazyMock`). Further, we are thinking of offering a `G::MockNoMem`, which instead of actually allocating the object would simply allocate inaccessible pages. This can be helpful in some cases where you can't properly construct an instance without having to mock away a lot of stuff. The fact that the memory is invalid then makes sure that you get at least well behaved test case termination when something accidentally tries to access the non-constructed object.

Another mock type that you may need is `G::StaticMock<MyLegacyStruct>()` which really is a static function and can be passed to Google Mock to expect on static member functions.

1.5 Working with Polymorphic Methods

Polymorphism requires some caution. There is one rule of thumb at the moment that you should memorize and speak in your dreams:

> *To mock a polymorphic call, you need to set your expectations on the method that will actually be called at runtime.*

Let's illustrate this with some examples. Assume we want to mock the following classes:

```
struct BaseA {
    virtual int inAOverriddenByB() { return 12; }
    virtual int inAOverriddenByBABC() { return 13; }
    virtual int inCImplementsAOverriddenByABC() = 0;
};

struct BaseB : public BaseA {
    virtual int inAOverriddenByB() { return 14; }
    virtual int inAOverriddenByBABC() { return 15; }
```

```
};

struct BaseC : public BaseA {
    virtual int inCImplementsAOverriddenByABC() { return 20; }
};

struct BaseABC : public BaseB, public BaseC {
    virtual int inCImplementsAOverriddenByABC() { return 23; }
    virtual int inAOverriddenByBABC() { return 24; }
};
```

The methods are named to reflect their polymorphic semantic and thus allows one to understand them without memorizing the class hierarchy (I hope you see the pattern). First, we will look at non-polymorphic mocking by just using an instance of `BaseA`:

```
G::MockFixture fixture;
G::Mock<BaseA> mock;

EXPECT_CALL(mock, inAOverriddenByB())
    .Times(1)
    .WillRepeatedly(Return(-1));

EXPECT_CALL(mock, inAOverriddenByBABC())
    .Times(1)
    .WillRepeatedly(Return(-2));

BaseA& instance = mock; // cast explicitly to not accidentally invoke GMock

ASSERT_EQ(-1, instance.inAOverriddenByB());
ASSERT_EQ(-2, instance.inAOverriddenByBABC());
```

In such a case, `BaseA` implements the polymorphic method that is called at runtime and the code works as intended.

Now let's do the same with an instance of `BaseB`.

```
G::MockFixture fixture;
G::Mock<BaseB> mock;

EXPECT_CALL(mock, inAOverriddenByB())
    .Times(1)
    .WillRepeatedly(Return(-3));

EXPECT_CALL(mock, inAOverriddenByBABC())
    .Times(1)
    .WillRepeatedly(Return(-4));

BaseB& instance = mock;

ASSERT_EQ(-3, instance.inAOverriddenByB());
ASSERT_EQ(-4, instance.inAOverriddenByBABC());
```

The same goes here. From a compiler's perspective, it doesn't really know (except sometimes) that, at compile-time, the methods of class `BaseB` are called. It will emit, same as before, a virtual call invocation through a vtable. Here again we mock the implementation that is called at runtime.

Now as a flawed example, let's see how you shouldn't do it:

```
G::MockFixture fixture;
G::LazyMock<BaseA> lazyA;

EXPECT_CALL(lazyA, inAOverriddenByB())
    .Times(2)
    .WillRepeatedly(Return(-3));

EXPECT_CALL(lazyA, inAOverriddenByBABC())
    .Times(2)
    .WillRepeatedly(Return(-4));

BaseB instanceB;
ASSERT_EQ(-3, instanceB.inAOverriddenByB());
ASSERT_EQ(-4, instanceB.inAOverriddenByBABC());

BaseA& instance = instanceB;
ASSERT_EQ(-3, instanceA.inAOverriddenByB());
ASSERT_EQ(-4, instanceA.inAOverriddenByBABC());
```

Here we mock the implementations provided by `BaseA` and then create an instance of a more derived class `BaseB`. This won't work. LegacyMock does mock the implementations of `BaseA` but they are never invoked (since their addresses in the vtable are overwritten by the methods in `BaseB`), so our expectations won't be satisfied. `lazyA` will not even be popped out of the queue here, because LegacyMock doesn't know that a more derived instance was created (there is no awareness of this kind available when LegacyMock parses header files).

As you may have noticed, so far we always re-implemented all methods in more derived classes. This is perfect, because it makes mocking simple. You have to be much more careful, when this is not the case. Let's look at some examples…

If we try to mock an instance of `BaseC`, then we will run into problems already:

[TBD]

To fix this, we need to use two mocks, since two different classes provide implementations of the methods we want to mock:

```
G::MockFixture fixture;
G::Mock<BaseC> mockC;
G::LazyMock<BaseA> lazyA;

EXPECT_CALL(lazyA, inAOverriddenByB())
    .Times(1)
    .WillRepeatedly(Return(-5));
```

```
EXPECT_CALL(lazyA, inAOverriddenByBABC())
    .Times(1)
    .WillRepeatedly(Return(-6));

EXPECT_CALL(mockC, inCImplementsAOverriddenByABC())
    .Times(1)
    .WillRepeatedly(Return(-7));

BaseA& instance = mockC;

// the following binds lazyA to the backing object of mockC
ASSERT_EQ(-5, instance.inAOverriddenByB());
ASSERT_EQ(-6, instance.inAOverriddenByBABC());
ASSERT_EQ(-7, instance.inCImplementsAOverriddenByABC());
```

In future, there might be a way to kind of *merge* mocks into one (thus freeing us from the task of dealing with several mocks for one single object instance), but we have no clean concept on how to do this at the moment, without introducing more problems than it solves.

1.6 Caution with Multi-Inheritance

[TBD: Depends on item SJFR-1535]

In short: It is supported but not yet stable or final (it's a hack).

1.7 The Code Generator

Almost anything accessible via `G::` is automatically generated code. Given a normal C++ header, LegacyMock uses a code generator, written in C++ and partially Python, to create the mocking infrastructure for a specific class of yours. It is important to get acquainted with this generator, since it is the primary tool not only for adding mocks for existing classes, but also to influence the feature set and nature of those mocks.

In "1.16 All Command Line Arguments for Mock-Generator", all command line arguments of the generator are documented. For now, we just want to give you an idea of what this generator does with a given header file:

```
/Path/To/SomeClass.h

namespace MockMe {
    struct SomeClassA {
        int someFunction();
        static int someFunction();
    };
}
```

```
struct SomeClassB {
    int someFunction();
    static int someFunction();
};
```

Now let's say we want to mock `MockMe::SomeClassA` declared in that header using LegacyMock. The first step is to setup a generator script, since usually you won't be mocking just one class:

```
from subprocess import call
import os, sys

pythonExe = sys.executable
currDir = os.path.dirname(os.path.abspath(__file__)) + "/"
cpp_to_mock = "/my/path/to/LegacyMock/citrix.mocking/src/cpp_to_mock.py"

call([pythonExe, cpp_to_mock,
          "-i", currDir + "AllInOne.h",
          "-o", currDir + "../GeneratedMocks/",
          "-c", "AllInOne",
          "-inc", "ThingsToMock/AllInOne.h",
          "--enable-tracing",
          "--enable-default-impl",
])
```

This script invokes the real generator under the hood for each mock you want to generate. The idea is to design this script invariant to its working directory and parameters, so it can easily be called from anywhere, anytime.

The actual call to the code generator reads in the header file, parses it into some sort of JSON syntax tree, and finally generates two C++ files (mock header and implementation) into the given output directory.

Those two files contain everything that is necessary to use the `G::` magic for `SomeClassA`. The CPP file obviously needs to be linked with your UnitTest executable. The header file needs to be included wherever you intend to use the mock. Please don't use any of the classes inside generated files directly, unless recommended to do so in this guide. Otherwise you might be subject to random breakage of your build when updating LegacyMock.

In the following sections, we will continue to explore advanced options that significantly influence the capabilities of the generated mocks.

1.8 Mocking Abstract Classes

To obtain virtual method addresses (which is required for mocking), LegacyMock needs to create an instance of the class in question. That is not possible for abstract classes though. To workaround the issue of instantiating abstract classes, you need to create a stub class by hand, which shall provide a default implementation for all abstract methods. Be sure not to override any already implemented virtual methods though, otherwise the resulting mock will misbehave in the sense that mocks for such overridden functions will never be called. If you look back at the polymorphism examples, `BaseA` was an abstract class that we mocked. The stub for implementing its abstract methods looks like this:

```
namespace internal {
    struct BaseA_ProbingInstance : public BaseA {
        virtual int inCImplementsAOverriddenByABC(int a) { return 0; }
    };
}
```

The only thing left is making LegacyMock aware of this stub, a so called “Probing instance” (used to probe for virtual method addresses) for `BaseA`. This needs to be done in the python generator. There is a special command line argument:

```
--set-probing-instance ::MockMe::internal::BaseA_ProbingInstance
```

It expects a fully qualified class name and this is the one that will be used to create the probing instance for the class you are trying to mock.

Keep in mind though, that it is not possible to mock pure virtual functions and they will be excluded automatically. Only the functions actually implemented by a particular class will get mocked.

1.9 Using Your Own Custom Mocks (Not Google Mock)

Google Mock is an optional component that can be turned off during mock generation by passing

```
--disable-gmock
```

to the code generator. It is just a plugin, one that is shipped with the system, but you can integrate your own mocks in very much the same way. For instance, call-tracing is implemented as plugin in too which, in contrast to GMock, is disabled by default and needs to be explicitly enabled via:

```
--enable-tracing
```

The core of the plugin system is a convenience layer, providing the features you already witnessed, namely everything following the magical “`G`::”. So how did we get there, when taking Google Mock as an example of a plugin we wanted to add (suppose it wasn’t already there)? In the header file

```
#include "../citrix.mocking/include/citrix/mocking/MockWrapper.h"
```

There is a policy template called `MockWrapper` that allows you to specialize the same foundation used for GMock for your very own mock implementations. It expects a template as template parameter. GMock uses the following policy for specialization:

```
#ifndef _CITRIX_MOCKING_GMOCK_H_
#define _CITRIX_MOCKING_GMOCK_H_

#include "../citrix.mocking/include/citrix/mocking/MockWrapper.h"

namespace citrix { namespace mocking {

template<class T>
class GMockContext {
public:
    BOOST_STATIC_ASSERT_MSG(
        ::citrix::mocking::internal::MockTraits<T>::hasGMock,
        "Google Mock implementations for class <T> could not be found");

    typedef typename internal::MockTraits<T>::TGoogleStaticMock TStatic;
    typedef typename internal::MockTraits<T>::TGoogleMockInstance TInstance;
};

typedef MockWrapper<GMockContext> GMock;

}}

#endif
```

The gist of it is for you to define the two types `TStatic` and `TInstance`, based on the template parameter `T` passed to your policy. This parameter will be the class a user wants to mock. Afterwards, you can alias the specialized `MockWrapper` to something more memorable, like `GMock`. Now, if someone wants to utilize your policy, she just needs to `GMock::MockFixture` on it. For further convenience, we aliased `GMock` with `G` in all examples within this guide.

The types you set for `TStatic` and `TInstance` need to obey certain rules. Let's look specifically at an example for `T` being `BaseA`. They should virtually inherit from `BaseAStaticMock` and `BaseAMockInstance`, which both are created by the code generator. You can then provide implementations for all virtual methods you have inherited and potentially add your own too. If you want to integrate your own mocks into code generation, you may find `internal::MockTraits<T>` helpful to get access to class specific, generated types.

1.10 Tracing C++ Calls

[TBD]

1.11 Creating a Custom Default-Instance-Allocator

There are various places inside LegacyMock where it needs to allocate instances of objects that are unknown to it. For instance, parameters passed to a mock object, default return values and also the class to mock itself. Now, normally all of that happens under the hood and you will notice nothing about it. But as soon as object construction gets complicated, parameters or return values have no default constructor,

or you want to alter the way instances are created, then you may have to special-case object construction. To make this possible, all such object constructions are routed through a function template which you can specialize if needed. Let's look at some examples.

```
::citrix::mocking::internal::allocateDefaultInstance<int64_t>()
```

[TBD]

1.12 Mocking Windows API

LegacyMock supports hooking of free api, like the Windows API, but basically any function defined outside of a namespace. You get the same benefits as with normal class mocks. All the features are available too, in the very same way. This works by simply creating fake classes for API functions you'd like to mock, for instance `RegOpenKeyExW`:

```
class Win32Registry {
public:
    static LSTATUS
        APIENTRY
        RegOpenKeyExW (
            HKEY hKey,
            LPCWSTR lpSubKey,
            DWORD ulOptions,
            REGSAM samDesired,
            PHKEY phkResult
        );

    static LSTATUS
        APIENTRY
        RegCloseKey (
            HKEY hKey
        );
};
```

You can group multiple related functions in the same class. For instance, all registry related Windows API functions could be mocked through the same class.

Then you can tell the code generator via

```
--mock-free-api
```

to link the mocks to free functions (instead of class members) with the same signatures as your static class members. For that to work, you need to make sure that those free functions are pre-declared inside the scope of the mock implementation. The recommended way to do this, is simply including all dependencies

(here it would be `<windows.h>`) into the file defining the class to be mocked. In our case, the file containing `Win32Registry`.

As done in the above example, it is recommended to only declare those static members, instead of providing a method body, so that if something goes wrong or you forget to tell LegacyMock that you want to hook free API, you will at least get a link-time error.

Now you can mock Windows API in the same manner as normal static class member functions:

```
EXPECT_CALL(G::StaticMock<Win32Registry>(), RegOpenKeyExW(
    HKEY_CURRENT_USER,
    StrEq(pluginRegPath),
    0,
    KEY_READ,
    _))
.Times(1)
.WillRepeatedly(DoAll(
    SetArgPointee<4>((HKEY)0x123),
    Return(0)
));

EXPECT_CALL(G::StaticMock<Win32Registry>(), RegCloseKey((HKEY)0x123))
    .Times(1)
    .WillRepeatedly(Return(0));
```

1.13 Accessing Protected & Private Members

It's easy to see the benefit of mocking or testing protected members, as they can be used by virtually anyone without much effort and thus should be tested as throughout as any public method. But even mocking private members is important (but you should never test them), as we will see in this section.

To add this capability to a mock, you first need to make sure that LegacyMock can read private & protected members by adding the following friend declaration into the class to mock:

```
template<typename> friend class ::CitrixMockingAccessor;
```

And you need to tell the code generator to mock private and/or protected members, which can be done by passing:

```
--mock-protected
--mock-private
```

This will immediately allow you to mock private & protected functions in the same way as public functions. There is no difference at all. They are virtually propagated into public scope.

So let's rather look at the more complicated part, which is *calling* a private or protected function.

Invocation of a private method `somePrivateFunction` on an `instance` of an object of type `MyClassToMock` looks like this:

```
G::Accessor<MyClassToMock>::somePrivateFunction(&instance, param1, param2)
```

There are two scenarios that can arise here. First, calling a private function while not having an active mock on the class. This will just do a usual function call, straightforward.

The second case is the more interesting one. Calling a function while having an active mock on that class. You certainly didn't intend to call on the mock itself. So LegacyMock prevents that, if you call a function (doesn't matter if public, protected or private) through `G::Accessor`, then the call will always invoke the actual implementation and never ever invoke a mock of this function. Be aware though that this is only true for this single function. If the function `A` you call through `G::Accessor` happens to invoke another function `B` of the same class, and that class has an active mock, then the mock for `B` is still invoked.

Even though all of this might sound confusing, it is actually an extremely powerful tool when it comes to testing legacy code. Basically, the common rule for writing UnitTests for legacy code is to only write tests for what you change. But if you have a class with 5000 lines of code that is interweaved all over the place, there is no way in hell this is going to happen. But if you could just test a single function at a time and mock away all other functions of the class, wouldn't that be great? `G::Accessor` allows you to do exactly this. Mocking away an entire class and then just running the implementation for a single specific function at a time. This way you can surgically dissect large legacy classes into little handy pieces and test them separately as you keep changing & evolving the legacy code base.

1.14 Accessing Member Variables

Unfortunately, legacy code often tends to violate fundamental rules of software design, testability is usually only the tip of the iceberg. One common cruelty is to read or write internal state that is either expected to be set by callers or queried by subsequent code after the call has returned, making functions that could easily have been stateless, stateful. To test such code, one must be able to read & write protected member variables and sometimes even private ones.

To add this capability to a mock, you first need to make sure that LegacyMock can read private & protected members by adding the following friend declaration into the class to mock:

```
template<typename> friend class ::CitrixMockingAccessor;
```

Then you need to tell the code generator to emit variable accessors:

```
--variable-accessors
```

And you need to tell it (optionally) to mock private and/or protected members, which can be done by passing:

```
--mock-protected
--mock-private
```

Now you can obtain references to any member variable via:

```
G::Accessor<MyClassToMock>::myPrivateVariable(&instance)
```

For instance variables, you have to provide the object `instance` you want to query the variable reference from. This parameter must be omitted for static member variables. The function name is always the same as the member variable name you want to access.

1.15 Mocking Constructors and Destructors

[TDB: Depends on item SJFR-1447]

Possible to get addresses of constructors and destructors with LegacyMock, but it's not integrated into mock generation yet.

1.16 All Command Line Arguments for Mock-Generator

This is basically what you get by passing `--help` to the generator.

```
--input "/path/to/some/file.h"
      Mandatory C++ header file to parse.

--outdir "/path/to/some/dir/"
      Mandatory output directory. All generated files will be stored in here. If the directory does not
      exist, it will be created.

--class-name "MyClassName"
      Process the class with the given name (unqualified, no namespaces). By default, the first class
      encountered in the header file is processed.

--class-prefix "typedef Whatever Whatever2;"
      Allows you to insert custom code into the public preamble of a class. This code is inserted into
      the public section before any generated code. It is also emitted into the global namespace
      within the mock implementation.

--include "Some/Include/File.h"
```

Include this file in the generated public mock header. Use multiple arguments to add multiple files.

`--enable-tracing`
Emit mocks that are tracable. This will likely require you to implement translators for the various types that might need to be serialized.

`--enable-default-impl`
By default, an abstract mock interface is emitted. But if you know for sure that all return types have default constructors, you can use this option to generate default implementations instead.

`--mock-free-api`
Interpret member functions as free functions (like Windows API). This will cause the free functions to be hooked but still grouped into mock classes with support for custom mocks, Google Mock & tracing as usual.

`--set-probing-instance "::some::namespace::MyProbingClass"`
Sets the name of the probing class (fully qualified, make sure you also add it's header to the include list). You need to provide an explicit probing class if the mocked class is abstract. A probing class should derive from the abstract class you want to mock and provide a default implementation for all abstract functions.

`--pure-static-class`
Must be specified if your class is not constructible. This will prevent any non-static members from being hooked.

`--mock-protected`
Also mock protected member functions. This requires you to change the source code of the target class to include "template<typename> friend class ::CitrixMockingAccessor;" in the class definition.

`--mock-private`
Also mock private member functions. This requires you to change the source code of the target class to include "template<typename> friend class ::CitrixMockingAccessor;" in the class definition.

`--variable-accessors`
Generates setters and getters for member variables, so you can assert on object state.

`--disable-gmock`
Do not generate any GMock code. This will significantly decrease compile times for generated headers and should be used when you don't intend to use GMock for the mocked class.

`--mock-lifetime`
Mock constructors & destructor. This is only supported for classes who have user defined constructors & destructor and will raise an error otherwise.

1.17 LegacyMock Cheat Sheet

To mock a polymorphic call, you need to set your expectations on the method that will actually be called at runtime.

`G`

A type alias for `GMock`, which is a type alias for `MockWrapper<GMockContext>`. You need to define `G` yourself. Only `GMock` is provided by LegacyMock when including `"../citrix.mocking/include/citrix/mocking/GMock.h"`.

`G::MockFixture`

An instance of this type should always life for the entire testcase and before you use any other LegacyMock features. You must not reuse a fixture for more than one `TEST`().

`G::Mock<MyClassToMock>`

A mock for the given class `MyClassToMock`, that also contains an instance of `MyClassToMock`. This instance is constructed immediately. Optional constructor arguments are forwarded to the constructor of `MyClassToMock`. You can only mock instance members with this.

`G::DefaultMock<MyClassToMock>`

A mock for the given class `MyClassToMock`, that handles all mock invocations that aren't handled by any other mock type. For instance, anything that is already bound to a `G::Mock<MyClassToMock>` won't be intercepted by a default mock. But other instances that currently have no associated mock, will be.

`G::LazyMock<MyClassToMock>`

A mock for the given class `MyClassToMock`, that does NOT act as an instance of. You can use it for GMock and that's it. It will enter a FIFO queue of lazy mocks and the topmost will be popped and assigned whenever a new instance of is created. This instance then is tied to the GMock expectations you set on it. You can only mock instance members with this.

`G::StaticMock<MyClassToMock>()`

A mock for all static members of class `MyClassToMock`. This expression, in contrast, is a function, not a type! This one function must be used for all calls, there is no way to use multiple mock instances for static members at the moment. You can only mock static members with this.

`G::Accessor<MyClassToMock>`

Provides access to all members, variables, functions, static and instance, of `MyClassToMock`. Calling a method through an accessor will call the original implementation and bypass all mocks, always.

```
template<typename> friend class ::CitrixMockingAccessor;
```

Add this to a production class you want to mock, so LegacyMock can access protected & private members.

`::citrix::mocking::internal::allocateDefaultInstance`

Specialize this function for a production class you want to mock, thereby providing your custom default instance.

2 How to integrate it into my project?

2.1 An Example Project

Let's look at a minimal project using LegacyMock to mock a method in a class and GMock & GTest to actually write a proper test. In the following, `$(LGM)` refers to the root of LegacyMock. You can find the source code in the `$(LGM)/examples/minimal` folder after obtaining LegacyMock as described above.

In the following we will briefly look at each involved file. Let's start with the class that's going to be mocked:

```
// File: MyClass.h

#include <string>

namespace LegacyDemo {

    class MyClass {
    public:
        int multiply(int a, int b);
    };

}
```

It has a function that multiplies both arguments and returns the result:

```
// File: MyClass.cpp

#include "MyClass.h"

#include <iostream>

namespace LegacyDemo {

    int MyClass::multiply(int a, int b) {
        return a * b;
    }

}
```

The crucial part now is to use the code generator to create a mock for it. Here, this is a bit much configuration overhead, but normally you will have a lot more classes and then each additional mocked class will only have minimal further overhead. You may want to use a python script for that, since python is needed anyway for code generation. The following script provides a "call" function which is a simple wrapper around invoking the code generator and allows you to still pass all parameters you want. All further mocks can be created by adding additional `call` invocations.

```
// File: GenerateMocks.py

import os, sys, subprocess

# make this script invariant to the current working directory
currDir = os.path.dirname(os.path.abspath(__file__)) + "/"
lgmRoot = currDir + "../../"
generatorExe = lgmRoot + "bin/citrix.mocking.generator.exe"

if not os.path.exists(currDir + "GeneratedMocks/"):
   os.makedirs(currDir + "GeneratedMocks/")

def call(args):
   args = [generatorExe,
      "--python", sys.executable,
      "--parser-script", lgmRoot + "citrix.mocking/src/CppHeaderParserStub.py",
   ] + args;

   if(subprocess.call(args) != 0):
      print "[FATAL]: Failed to generate mock for " + str(args)
      sys.exit(-1)

# this is the important part
call([
   "--input", currDir + "MyClass.h",
   "--outdir", currDir + "GeneratedMocks/",
   "--class-name", "MyClass",
   "--include", "MyClass.h",
   "--enable-default-impl",
   "--safe-update-check",
])
```

Note that this script directly references files inside `$(LGM)`, so if you create your own application using LegacyMock, then you also need to adjust the `lgmRoot` variable to match your folder structure. We will later look at when to run this script, but this is all you need to do for generating mocks.

In the main file, we need to setup Google Mock and also have the one and only test-case in the project:

```
// File: main.cpp

#include "gtest/gtest.h"
#include "gmock/gmock.h"

#include "../citrix.mocking/include/citrix/mocking/GMock.h"
#include "GeneratedMocks/MyClassMock.h"

typedef citrix::mocking::GMock G; // a shorthand
```

```
using namespace LegacyDemo;

TEST(MyClassTest, test) {

    G::MockFixture fixture;

    EXPECT_CALL(G::DefaultMock<MyClass>(), multiply(3, 4)).
        WillOnce(testing::Return(-1));

    MyClass someInstance;

    // if the original function would be invoked, this assertion would fail...
    ASSERT_EQ(-1, someInstance.multiply(3, 4));
}


int main(int argc, char** argv) {
    testing::InitGoogleMock(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The guide should explain all of what's in here.

The question now is how to build the thing. We added a simple CMake script for that. It requires Python and Boost to be installed (but doesn't need any Boost binaries). It runs the above python script whenever you configure CMake and also runs it before every build. This makes sure your mocks are always up-to-date. Further it adds `$(LGM)/include` as additional include directory, which is necessary for the compiler to resolve all header files implicitly pulled in by using LegacyMock.

```
// File: CMakeLists.txt

cmake_minimum_required (VERSION 2.8)
project (LegacyDemo)

find_package(PythonInterp)
find_package(Boost)

execute_process(COMMAND "${PYTHON_EXECUTABLE}"
"${CMAKE_CURRENT_SOURCE_DIR}/GenerateMocks.py")

add_executable (LegacyDemo

# our little program
    main.cpp
    MyClass.cpp
    MyClass.h

# this will compile all of LegacyMock right into our program
    ../../include/citrix/srclink/citrix.mocking-all.cpp

# this will compile the generated mock
    GeneratedMocks/MyClassMock.cpp
    GeneratedMocks/MyClassMock.h)
```

```
target_include_directories (LegacyDemo PUBLIC
    "${CMAKE_CURRENT_SOURCE_DIR}"

# LegacyMock requires you to add this to includes atm
    "${Boost_INCLUDE_DIRS}"
    "${CMAKE_CURRENT_SOURCE_DIR}/../../include"

# This project also generates GMocks for the test-class so we need
# to include GMock.
    "${CMAKE_CURRENT_SOURCE_DIR}/../../citrix.gmock/include"
)

add_custom_command(
    TARGET LegacyDemo
    PRE_BUILD
    COMMAND "${PYTHON_EXECUTABLE}" "${CMAKE_CURRENT_SOURCE_DIR}/GenerateMocks.py"
)
```

Now to build the project, you can just start CMake, enter to path to the example folder (`$(LGM)/examples/minimal/`), chose a build directory and press "Configure" & "Generate". After that, you will find a Visual Studio solution inside your build directory.

Congrats!

2.2 Massaging your Header Files

[TBD]

2.3 Compiling LegacyMock

Unless you know what you are doing you should do the first compilation via `$(LGM)/build.py`. This script will properly setup CMake to generate a valid project for all the tons of different supported configurations. The build script will yell at you when you do something wrong, so just try to use it and maybe read its command line help.

2.4 Selecting Google Mock & Test

At the moment it is recommended to stick with "citrix.gmock/include" which comes with LegacyMock (GMock 1.7 nightly)

[TBD]