CMock: An Introduction

What Exactly Are We Talking About Here?

CMock is a nice little tool which takes your header files and creates a mock interface for it so that you can more easily unit test modules that touch other modules. It goes something like this: For each function prototype in your header, like this one:

```
int DoesSomething(int a, int b);
```

...you get an automatically generated DoesSomething function that you can link to *instead* of your real DoesSomething function. By using this mocked version, you can then verify that it receives the data you want, and make it return whatever data you desire, make it throw errors when you want, and more... Create these for everything your latest real module touches, and you're suddenly in a position of power: You can control and verify every detail of your latest creation.

To make that easier, CMock also gives you a bunch of functions like the ones below, so you can tell that generated DoesSomething function how to behave for each test:

```
void DoesSomething_ExpectAndReturn(int a, int b, int toReturn);
void DoesSomething_ExpectAndThrow(int a, int b, EXCEPTION_T error);
void DoesSomething_StubWithCallback(CMOCK_DoesSomething_CALLBACK YourCallback);
void DoesSomething_IgnoreAndReturn(int toReturn);
```

You can pile a bunch of these back to back, and it remembers what you wanted to pass when, like so:

```
test_CallsDoesSomething_ShouldDoJustThat(void)
{
    DoesSomething_ExpectAndReturn(1,2,3);
    DoesSomething_ExpectAndReturn(4,5,6);
    DoesSomething_ExpectAndThrow(7,8, STATUS_ERROR_OOPS);

    CallsDoesSomething( );
}
```

This test will call CallsDoesSomething, which is the function we are testing. We are expecting that function to call DoesSomething three times. The first time, we check to make sure it's called as DoesSomething(1, 2) and we'll magically return a 3. The second time we check for DoesSomething(4, 5) and we'll return a 6. The third time we verify DoesSomething(7, 8) and we'll throw an error instead of returning anything. If CallsDoesSomething gets any of this wrong, it fails the test. It will fail if you didn't call DoesSomething enough, or too much, or with the wrong arguments, or in the wrong order.

CMock is based on Unity, which it uses for all internal testing. CMock is both a C file (with a couple headers) and a Ruby script which generates C and header files for you.

Expectations

In addition to the mocks themselves, CMock will generate the following functions for use in your tests. The _*Expect* functions are always generated. The other functions are only generated if those plugins are enabled.

Your basic staple _Expect functions will be used for most of your day-to-day CMock work. By calling this, you are telling CMock that you expect that function to be called during your test. It also specifies which arguments you expect it to be called with, and what return value you want returned when that happens. You can call this function multiple times back-to-back in order to queue up multiple calls.

```
void func(void) => void func_Expect(void)
void func(params) => void func_Expect(expected_params)
retval func(void) => void func_ExpectAndReturn(retval_to_return)
retval func(params) => void func_ExpectAndReturn(expected_params, retval_to_return)
```

ExpectAnyArgs

This behaves just like the _Expect calls, except that it doesn't really care what the arguments are that the mock gets called with. It still counts the number of times the mock is called and it still handles return values if there are some.

```
void func(void) => void func_ExpectAnyArgs(void)
void func(params) => void func_ExpectAnyArgs(void)
retval func(void) => void func_ExpectAnyArgsAndReturn(retval_to_return)
retval func(params) => void func_ExpectAnyArgsAndReturn(retval_to_return)
```

ExpectWithArray

An _ExpectWithArray is another variant of _Expect. Like _Expect, it cares about the number of times a mock is called, the arguments it is called with, and the values it is to return. This variant has another feature, though. For anything that resembles a pointer or array, it breaks the argument into TWO arguments. The first is the original pointer. The second specifies the number of elements it is to verify of that array. If you specify 1, it'll check one object. If 2, it'll assume your pointer is pointing at the first of two elements in an array. If you specify zero elements, it will check just the pointer if `:smart` mode is configured or fail if `:compare_data` is set.

```
void func(void) => (nothing. In fact, an additional function is only generated
                    if the params list contains pointers)
void func(ptr * param, other) => void func_ExpectWithArray(ptr* param,
                                                           int param_depth,
                                                           other)
retval func(void) => (nothing. In fact, an additional function is only generated
                      if the params list contains pointers)
retval func(other, ptr* param) => void func_ExpectWithArrayAndReturn(other,
                                                                     ptr* param,
                                                                     int param_depth,
                                                                     retval_to_return)
```

Ignore

Maybe you don't care about the number of times a particular function is called or the actual arguments it is called with. In that case, you want to use _Ignore. Ignore only needs to be called once per test. It will then ignore any further calls to that particular mock. The _IgnoreAndReturn works similarly, except that it has the added benefit of knowing what to return when that call happens. If the mock is called more times than IgnoreAndReturn was called, it will keep returning the last value without complaint. If it's called less times, it will also ignore that. You SAID you didn't care how many times it was called, right?

```
void func(void) => void func_Ignore(void)
void func(params) => void func_Ignore(void)
retval func(void) => void func_IgnoreAndReturn(retval_to_return)
retval func(params) => void func_IgnoreAndReturn(retval_to_return)
```

Ignore Arg

Maybe you overall want to use _Expect and its similar variations, but you don't care what is passed to a particular argument. This is particularly useful when that argument is a pointer to a value that is supposed to be filled in by the function. You don't want to use _ExpectAnyArgs, because you still care about the other arguments. Instead, before any of your _Expect calls are made, you can call this function. It tells CMock to ignore a particular argument for the rest of this test, for this mock function.

```
void func(params) => void func_IgnoreArg_paramName(void)
```

ReturnThruPtr

Another option which operates on a particular argument of a function is the _ReturnThruPtr plugin. For every argument that resembles a pointer or reference, CMock generates an instance of this function. Just as the AndReturn functions support injecting one or more return values into a queue, this function lets you specify one or more return values which are queued up and copied into the space being pointed at each time the mock is called.

```
void func(param1) => void func_ReturnThruPtr_paramName(val_to_return)
                  => void func_ReturnArrayThruPtr_paramName(cal_to_return, len)
                  => void func_ReturnMemThruPtr_paramName(val_to_return, size)
```

Callback

If all those other options don't work, and you really need to do something custom, you still have a choice. As soon as you stub a callback in a test, it will call the callback whenever the mock is encountered and return the retval returned from the callback (if any) instead of performing the usual expect checks. It can be configured to check the arguments first (like expects) or just jump directly to the callback.

```
void func(void) => void func_StubWithCallback(CMOCK_func_CALLBACK callback)
               where CMOCK_func_CALLBACKlooks like: void func(int NumCalls)

void func(params) => void func_StubWithCallback(CMOCK_func_CALLBACK callback)
               where CMOCK_func_CALLBACK looks like: void func(params, int NumCalls)

retval func(void) => void func_StubWithCallback(CMOCK_func_CALLBACK callback)
               where CMOCK_func_CALLBACK looks like: retval func(int NumCalls)

retval func(params) => void func_StubWithCallback(CMOCK_func_CALLBACK callback)
               where CMOCK_func_CALLBACK looks like: retval func(params, int NumCalls)
```

Cexception

Finally, if you are using Cexception for error handling, you can use this to throw errors from inside mocks. Like _Expects, it remembers which call was supposed to throw the error, and it still checks parameters first.

```
void func(void) => void func_ExpectAndThrow(value_to_throw)
void func(params) => void func_ExpectAndThrow(expected_params, value_to_throw)
retval func(void) => void func_ExpectAndThrow(value_to_throw)
retval func(params) => void func_ExpectAndThrow(expected_params, value_to_throw)
```

Installation

The first thing you need to do to install CMock is to get yourself a copy of Ruby. If you're on Linux or OS X, you probably already have it. You can prove it by typing the following:

```
ruby --version
```

If it replied in a way that implies ignorance, then you're going to need to install it. You can go to ruby-lang to get the latest version. You're also going to need to do that if it replied with a version that is older than 1.9.3… though to be honest, we'd really recommend staying above 2.0.0 because there are a number of improvements you'll miss out on if you're older than that. Go ahead. We'll wait.

Once you have Ruby, you have three options for grabbing CMock.

Git

Your first option is to clone the latest CMock repository directly from GitHub. This will let you update it easily later. If you're planning to put CMock into your repository with your source project and you're using Git for your release code already, you might consider assigning this project as a Git subproject.

```
git clone https://github.com/ThrowTheSwitch/CMock.git
git pull
```

Zip

The second option is to download the latest CMock zip from Github and the unzip it into your local project. You can find it at https://github.com/ThrowTheSwitch/CMock/archive/master.zip

Ceedling

Finally, you can choose to install Ceedling, *which has it built in!* Through your command shell you can install it by using the Ruby package manager, like so:

```
gem install ceedling
```

You'll want to open the Ceedling documentation for more information on this option.

Getting Started

If you've read the Unity documentation, you already know that each test gets compiled as its own executable. For a module Beaker.c, this means compiling the following files together:

- Beaker.c
- TestBeaker.c
- TestBeaker_Runner.c
- unity.c

When using CMock, there are a few more files that get compiled into the mix. The obvious one is "CMock.c". There are more, though. You'll also want to compile in the *mock* version of any files that are called by Beaker.c. Let's say Beaker.c usually includes Glassware.h and Measure.h. To test this module, you'd want to also compile and link the following:

- cmock.c
- MockGlassware.c
- MockMeasure.c

Before being able to compile and link, though, you had to first generate those files using CMock. Before that, you had to determine which files needed to be compiled together. This is probably the trickiest part to rolling your own test setup with mocks. It's the reason we created Ceedling. It's also the reason there is an example project in CMock itself. Both of these rely on the same convention, which is that the Test file is searched for includes. For any header file that has a matching C file, it assumed you want that C file to be part of your build. Following the example above, our test file would then look something like this:

```
#include "unity.h"
#include "cmock.h"
#include "Beaker.h"
#include "MockGlassware.h"
#include "MockMeasure.h"
```

The runner is always assumed by these tools, but the rest of the header files drive the test engine to know it needs to mock, compile, and link the corresponding C files.

You can read more details of this in the configuration guide, in articles on ThrowTheSwitch.org, and in the examples.