What the What?

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. 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, it should call DoesSomething(1, 2) and we'll magically return a 3. The second time it will call DoesSomething(4, 5) and we'll return a 6. The third time it will call 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 currently based on Unity, which is uses for all internal testing.

Generated Mock Module Summary

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:

Expect:

Your basic staple Expects which will be used for most of your day to day CMock work.

Original Function	Generated Mock Function
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)

Array:

An ExpectWithArray will check as many elements as you specify. If you specify zero elements, it will check just the pointer if :smart mode is configured or fail if :compare_data is set.

Original Function	Generated Mock Function
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)

Callback:

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.

Original Function	Generated Mock Function
void func(void)	void func_StubWithCallback(CMOCK_func_CALLBACK callback) where CMOCK_func_CALLBACK looks 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:

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.

Original Function	Generated Mock Function
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)

Ignore:

Calling this will force CMock to ignore calls to this function. It won't check parameters and you can specify multiple returns or a single value to always return, whichever you prefer.

Original Function	Generated Mock Function
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)

Running CMock

CMock is a Ruby script and class. You can therefore use it directly from the command line, or include it in your own scripts or rakefiles.

Mocking from the Command Line

After unpacking CMock, you will find CMock.rb in the 'lib' directory. This is the file that you want to run. It takes a list of header files to be mocked, as well as an optional yaml file for a more detailed configuration (see config options below).

For example, this will create three mocks using the configuration specified in MyConfig.yml:

```
ruby cmock.rb -oMyConfig.yml super.h duper.h awesome.h
```

And this will create two mocks using the default configuration:

```
ruby cmock.rb ../mocking/stuff/is/fun.h ../try/it/yourself.h
```

Mocking From Scripts or Rake

CMock can be used directly from your own scripts or from a rakefile. Start by including cmock.rb, then create an instance of CMock. When you create your instance, you may initialize it in one of three ways.

You may specify nothing, allowing it to run with default settings:

```
cmock = CMock.new
```

You may specify a YAML file containing the configuration options you desire:

```
cmock = CMock.new('../MyConfig.yml')
```

You may specify the options explicitly:

```
cmock = CMock.new('plugins' => ['cexception','ignore'],'mock_path' => 'my/mocks/')
```

Config Options:

The following configuration options can be specified in the yaml file or directly when instantiating.

Passed as Ruby, they look like this:

```
{ :attributes => [“__funky”, “__intrinsic”], :when_ptr => :compare }
```

Defined in the yaml file, they look more like this:

```
---
:cmock:
  :attributes:
    - __funky
    - __intrinsic
  :when_ptr: :compare
```

Option	**Purpose**
:attributes	These are attributes that CMock should ignore for you for testing purposes. Custom compiler extensions and externs are handy things to put here.
:cexception_include	Tell :cexception plugin where to find CException.h... only need to define if it's not in your build path already.
:enforce_strict_ordering	CMock always enforces the order that you call a particular function, so if you expect GrabNabber(int size) to be called three times, it will verify that the sizes are in the order you specified. You might also want to make sure that all different functions are called in a particular order. If so, set this to true.
:includes	An array of additional include files which should be added to the mocks. Useful for global types and definitions used in your project.
:memcmp_if_unknown	This is true by default. When true, CMock will just do a memory comparison of types that it doesn't recognize (not standard types, not in :treat_as, and not in a unity helper). If you instead want it to throw an error, just set this to false.
:mock_path	The directory where you would like the mock files generated to be placed.
:mock_prefix	The prefix to append to your mock files. Defaults to “Mock”, so a file “USART.h” will get a mock called “MockUSART.c”
:plugins	An array of which plugins to enable. 'expect' is always active. Also available currently are :ignore, :arrays, :cexception, and :callback
:treat_as	The :treat_as list is a shortcut for when you have created typedefs of standard types. Why create a custom unity helper for UINT16 when the unity function TEST_ASSERT_EQUAL_HEX16 will work just perfectly? Just add 'UINT16' => 'HEX16' to your list (actually, don't. We already did that one for you).
:treat_as_void	We've seen “fun” legacy systems typedef 'void' with a custom type, like MY_VOID. Add any instances of those to this list to help CMock understand how to deal with your code.
:unity_helper	If you have created a header with your own extensions to unity to handle your own types, you can set this argument to that path. CMock will then automagically pull in your helpers and use them. The only trick is that you make sure you follow the naming convention: TEST_ASSERT_EQUAL_YourType
:when_no_prototypes	When you give CMock a header file and ask it to create a mock out of it, it usually contains function prototypes (otherwise what was the point?). You can control what happens when this isn't true. You can set this to :warn, :ignore, or :error
:when_ptr	You can customize how CMock deals with pointers (c strings result in string comparisons... we're talking about other pointers here). Your options are :compare_ptr to just verify the pointers are the same, :compare_data or :smart to verify that the data is the same. :compare_data and :smart behaviors will change slightly based on if you have the array plugin enabled. By default, they compare a single element of what is being pointed to. So if you have a pointer to a struct called ORGANS_T, it will compare one ORGAN_T (no matter how big that is).