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, 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. It uses Ruby to do all the main work (versions 1.8.6 through 1.9.2).

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. It can be configured to check the arguments first (like expects) or just jump directly to the callback.

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:

This plugin supports two modes. You can use it to force CMock to ignore calls to specific functions or to just ignore the arguments passed to those functions. Either way 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
```

```
If you don't want to use YAML and want to inject your options directly to the command line, you can
specify most options directly using a --key=val notation.  Quotation marks around the value are
optional.  The key is always plain text.  The value is either a number, text, symbol, or array
separated by semicolons.  Try a few examples on:
```

```
ruby cmock.rb --mock_path=”build/mocks” blah.h
ruby cmock.rb --treat_externs=:include blah.h
ruby cmock.rb --attributes=”__irg;__fiq;register” blah.h
ruby cmock.rb --verbosity=1 blah.h
ruby cmock.rb --memcmp_if_unknown=false blah.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:

Passed as Ruby, configuration options look like this:

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

Defined in the yaml file, configuration options 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.
:callback_after_arg_check	Tell :callback plugin to do the normal argument checking before it calls the callback function. This defaults to false, where the callback function is called instead of the argument verification.
:callback_include_count	Tell :callback plugin to include an extra parameter to specify the number of times the callback has been called. If set to false, the callback has the same interface as the mocked function. This can be handy when you're wanting to use callback as a stub.
: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.
:framework	Currently the only option is :unity. Eventually if we support other unity test frameworks (or if you write one for us), they'll get added here.
:ignore	Tell :ignore plugin to ignore :args_only or :args_and_calls (default) where it doesn't even care how many times the mock was called
:includes	An array of additional include files which should be added to the mocks. Useful for global types and definitions used in your project. There are more specific versions if you care WHERE in the mock files the includes get placed. You can define any or all of :includes_h_pre_orig_header, :includes_h_post_orig_header, :includes_c_pre_header, :includes_c_post_header
: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, :array, :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). Maybe you have a type that is a pointer to an array of unsigned characters? No problem, just add 'UINT8_T*' => 'HEX8*'
: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.
:treat_externs	Set to :include to mock externed functions or :exclude to ignore them (the default).
:unity_helper_path	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: UNITY_TEST_ASSERT_EQUAL_YourType
:verbosity	0 for errors only. 1 for errors and warnings. 2 for normal. 3 for verbose
: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 ORGAN_T, it will compare one ORGAN_T (whatever that is).

Compiled Options:

A number of #defines also exist for customizing the cmock experience.

CMOCK_MEM_STATIC or CMOCK_MEM_DYNAMIC

Define one of these to determine if you want to dynamically add memory during tests as required from the heap. If static, you can control the total footprint of Cmock. If dynamic, you will need to make sure you make some heap space available for Cmock.

CMOCK_MEM_SIZE

In static mode this is the total amount of memory you are allocating to Cmock. In Dynamic mode this is the size of each chunk allocated at once (larger numbers grab more memory but require less mallocs).

CMOCK_MEM_ALIGN

The way to align your data to. Not everything is as flexible as a PC, as most embedded designers know. This defaults to 2, meaning align to the closest 2^2 → 4 bytes (32 bits). You can turn off alignment by setting 0, force alignment to the closest uint16 with 1 or even to the closest uint64 with 3.

CMOCK_MEM_PTR_AS_INT

This is used internally to hold pointers... it needs to be big enough. On most processors a pointer is the same as an unsigned long... but maybe that's not true for yours?

CMOCK_MEM_INDEX_TYPE

This needs to be something big enough to point anywhere in Cmock's memory space... usually it's an unsigned int.