OPMOCK 2.1 for C and C++

User Manual

Table of content

1 What is Opmock 2?

opmock 2 is a C and C++ stubbing, mocking, and unit testing framework. It allows developers to write tests for a single software layer, with good isolation of dependencies. The unit testing part can easily be replaced with another framework like Google test or Cpp Unit.

This way of testing, often called « unit testing » or “micro-testing”, should be familiar to people applying XP and TDD.

Opmock 2 is actually a C and C++ code generator written in C++, base on the clang/llvm tools.

Opmock 2 is currently only supported on Unix and Unix like systems. Versions for other systems/architectures may come one day provided that they support Clang.

2 Installing opmock 2

Opmock 2 is executed as a single binary that can be installed anywhere and does not require any administrative rights.

However, the current version does not come with a pre-built binary due to the large number of possible combinations with compilers and operating systems. You'll have to compile it yourself.

2.1 Compiling opmock 2 from the sources

To compile Opmock 2 from the sources, you need first to compile LLVM 3.2 and CLANG 3.2 from the sources. You can follow the extensive instructions from the CLANG/LLVM web site, or just use these condensed instructions :

Download the llvm/clang sources from:

http://llvm.org/releases/3.2/llvm-3.2.src.tar.gz

and

http://llvm.org/releases/3.2/clang-3.2.src.tar.gz

Create a folder and copy the 2 archives in it. In this section, we'll name the folder “foo” but you'll have to replace it by the name you've chosen.

You should hence have a tree like:

```
foo/
        llvm-3.2.src.tar.gz
        clang-3.2.src.tar.gz
```

unpack the llvm archive:

```
>cd foo
>tar xzf llvm-3.2.src.tar.gz
```

This creates a folder llvm-3.2.src. Rename it to 'llvm':

```
>mv llvm-3.2.src llvm
```

Now uncompress the clang archive:

```
>tar xzf clang-3.2.src
```

Similarly you rename it:

```
>mv clang-3.2.src clang
```

Then move it inside the llvm source tree:

```
>mv clang llvm/tools/
```

Configure the llvm source tree:

```
>cd llvm
>./configure --prefix=YOURPREFIX
```

Replace **YOURPREFIX** with the place where you want to install the compiled llvm/clang distribution. For example, /home/user/llvm32

If you want a minimal and fast build, then you may want to use:

```
>./configure --enable-targets=host-only --prefix=YOURPREFIX
```

Then compile llvm and clang (this takes some time, and typically requires GCC/G++ > 4.2):

```
>make -j 4
>make install
```

Now llvm/clang should be installed in $YOURPREFIX. You need to check out the opmock sources from subversion:

```
>svn checkout svn://svn.code.sf.net/p/opmock/code/ opmock-code
>cd opmock-code
>cd opmock2
```

Before compiling, you'll need to edit the Makefile so that it can find your clang/llvm installation. For example:

```
CXXFLAGS := $(shell ~/dev/llvm32_install/bin/llvm-config
--cxxflags) $(RTTIFLAG) -static -g
```

You want to replace **~/dev/llvm32_install** by the path where you installed llvm and clang (YOURPREFIX). Then save the makefile.

```
>make
```

This will produce an opmock2 binary in the current path. You may want to strip it to reduce its size:

```
>strip opmock2
```

Now you can copy the opmock2 executable in any place you like. Be sure to copy as well the **support** folder and all its content, as you'll need it for code generation. The easy option is to install both the opmock2 binary and the support folder next to each other.

3 Opmock tutorial for C

This section presents opmock 2 in a very progressive way. If you are already familiar with TDD and mocking, you may want to jump directly to advanced sections like section 3.

Please note that in the samples folder in the opmock sources, you can find plenty of working examples. They are, by the way, the regression test suite for Opmock.

3.1 Micro-testing : Fizzbuzz!

We want to implement the Fizzbuzz game. The rules are simple:

- If I get a number which is a multiple of '3' then I answer “FIZZ”
- If I get a number which is a multiple of '5' then I answer “BUZZ”
- If I get a number which is both a multiple of '3' and '5' I answer “FIZZBUZZ”
- If I get a number which is neither a multiple of '3' or '5' I answer “NA”

That's a nice isolated algorithm, simple enough to have no dependencies on additional functions, classes or libraries. We want to write micro tests for it. The best way would be to write these tests first, in a TDD fashion. In this documentation, we will just show both code and tests at the same time, but keep in mind that the tests have been written first, in the typical red-green cycle of TDD.

Note : this example makes use of the unit testing framework coming with Opmock. You can perfectly use another framework and use only the mocking capabilities.

3.1.1 Getting started : the build system

The first thing we need is a working build system. We base our example on make. Users of other environments will have to adapt what follows.

Here's the content of our makefile:

```
1  CPPFLAGS=-O0  -ggdb
2  OBJECTS = fizzbuzz.o fizzbuzz_test.o main.o opmock.o
3
4  all: fizzbuzzTest
5     ./fizzbuzz_test
6
7  fizzbuzz.o: fizzbuzz.h
8  fizzbuzz_test.o: fizzbuzz.h
9  opmock.o: opmock.h
10
11 fizzbuzzTest: $(OBJECTS) fizzbuzz.h
12    gcc -o fizzbuzz_test $(OBJECTS)
13
14 clean:
15    -rm -f $(OBJECTS)
16    -rm -f fizzbuzz_test
```

On line 1 we define some flags for the C compiler. They're not mandatory, but setting the optimizations to level 0 help debugging.

On line 2 we set the list of object files in the build. We have fizzbuzz.o for the code itself, fizzbuzz_test.o for the tests, and opmock.o for the unit testing framework and some support code. This part of the framework is delivered as a single source file, opmock.c (in the folder “support” of the binary distribution). It can be either compiled in each build, or compiled as a library and shared. Your choice.

The 'all' target will at the same time compile a test executable, and run it to get the tests results.

Line 7 to line 12 we give some dependencies rules to recompile C files if the header files they depend on are touched.

On line 14 we define a 'clean' target, removing all objects files.

Now that we have a working build, let's write some tests.

3.1.2 Writing micro tests

Let's have a look to the fizzbuzz_test.c file:

```
#include "fizzbuzz_test.h"
```

```
#include "fizzbuzz.h "
#include "opmock.h"
#include <stdlib.h>

void test_fizzbuzz_with_3()
{
  char *res = fizzbuzz(3);
  OP_ASSERT_EQUAL_CSTRING("FIZZ", res);
  free(res);
}

void test_fizzbuzz_with_5()
{
  char *res = fizzbuzz(5);
  OP_ASSERT_EQUAL_CSTRING("BUZZ", res);
  free(res);
}

void test_fizzbuzz_with_15()
{
  char *res = fizzbuzz(15);
  OP_ASSERT_EQUAL_CSTRING("FIZZBUZZ", res);
  free(res);
}

void test_fizzbuzz_many_3()
{
  int i;
  for(i = 1; i < 1000; i++) {
    if((i % 3 == 0) && ((i % 5) != 0)) {
      char *res = fizzbuzz(i);
      OP_ASSERT_EQUAL_CSTRING("FIZZ", res);
      free(res);
    }
  }
}

void test_fizzbuzz_many_5()
{
  int i;
  for(i = 1; i < 1000; i++) {
    if((i % 3 != 0) && ((i % 5) == 0)) {
      char *res = fizzbuzz(i);
      OP_ASSERT_EQUAL_CSTRING("BUZZ", res);
```

```
      free(res);
    }
  }
}

void test_fizzbuzz_many_3_and_5()
{
  int i;
  for(i = 1; i < 1000; i++) {
    if((i % 3 == 0) && ((i % 5) == 0)) {
      char *res = fizzbuzz(i);
      OP_ASSERT_EQUAL_CSTRING("FIZZBUZZ", res);
      free(res);
    }
  }
}
```

All tests in this file follow the same simple pattern:

- A micro test is a function that takes no parameters and returns nothing
- The function name is free; my personal convention is to name the test test_functionName_testDescription, but you can choose whatever suits you. *(note that if you want to use the helper script to generate automatically the list of tests, you shall follow this convention. Have a look to the advanced section).*
- A micro test is made of 3 phases : set input parameters, call the function or class under test, check the result. In addition, if you use mocks, you can have a verify phase where you check that your dependencies were called properly.

Opmock provides you some macros to ease testing, like

```
OP_ASSERT_EQUAL_CSTRING(expected value, actual value);
```

The complete list of OP_ASSERT* macros can be found in the file `opmock.h`. They all take the expected value as first parameter, and the actual value as second parameter.

If a macro fails, it will increment a global error counter, then return from the current scope (a function or a class method). That's the reason why you should use these macros only in void(void) functions.

3.1.3 Writing the implementation

One possible implementation of the fizzbuzz function is:

```
#include "fizzbuzz.h"

#include <string.h>
```

```
#include <stdlib.h>
#include <stdio.h>

char * fizzbuzz (int i)
{
  char *result = (char *) calloc(1, 20);

  if (!(i % 3)) strcpy(result, "FIZZ");
  if (!(i % 5)) strcat(result, "BUZZ");

  if(!strlen(result)) sprintf(result, "%d", i);
  return result;
}
```

A very simple function indeed, but you are more interested in tests than actual code, don't you?

3.1.4 Calling the tests

When these tests are written, we can run them just by calling the functions, or better, we can register them with opmock. This is done in the main.c file:

```
#include "opmock.h"
#include "fizzbuzz_test.h"

int main(int argc, char *argv[])
{
  opmock_test_suite_reset();
  opmock_register_test(test_fizzbuzz_with_3, "test_fizzbuzz_with_3");
  opmock_register_test(test_fizzbuzz_with_5, "test_fizzbuzz_with_5");
  opmock_register_test(test_fizzbuzz_with_15, "test_fizzbuzz_with_15");
  opmock_register_test(test_fizzbuzz_many_3, "test_fizzbuzz_many_3");
  opmock_register_test(test_fizzbuzz_many_5, "test_fizzbuzz_many_5");
  opmock_register_test(test_fizzbuzz_many_3_and_5,
"test_fizzbuzz_many_3_and_5");
  opmock_test_suite_run();
  return 0;
}
```

All tests you want to run must be registered against opmock using opmock_register_test (provided that you want to use opmock test report).

Now, if you type:

```
>make
```

You should get this report:

```
OK test 'test_fizzbuzz_with_3'
OK test 'test_fizzbuzz_with_5'
OK test 'test_fizzbuzz_with_15'
OK test 'test_fizzbuzz_many_3'
OK test 'test_fizzbuzz_many_5'
OK test 'test_fizzbuzz_many_3_and_5'
OPMOCK : 6 tests run, 0 tests failed.
```

Note also that you will get a colored output, with red for errors and green when everything is ok.

3.2 Using a mock

3.2.1 Why mocking?

Writing micro tests is easy. At least, it is in the simple example given above. However, in real code, you often have to deal with dependencies. For example, you develop something, but you must integrate a 3rd party library. This 3rd party library is not available yet – but you want to start coding against its interface.

Another situation is when you're developing a module, and it depends on an existing module. In the scope of your micro tests, you don't want to bring in all the dependencies of the additional module, because it's too complex and requires an access to a specific hardware which is available only in the lab. You want to cut the dependencies chain as short as possible.

The traditional way of dealing with this is to stub your dependencies. That is, writing a “fake” implementation of the dependency and link with it. This works well, but:

- It's usually very tedious to write stubs manually
- With C and C++, you can have only a single stub implementation in the same build. If you want to switch the stub implementation at runtime, you have to fall back to dirty tricks like global variables so that the stub knows how to answer. The only other way would be to have multiple builds, each one with a different stub implementation.
- Your stub file “life cycle” is separate of your tests life cycle
- Your stub usually does not record the parameters nor the number of times it was called (this is what a mock would do, to allow a verify phase)

In this situation, automatically generating stubs and/or mocks saves a lot of time. Opmock can do it for you, generating mocks that fully replace the original implementation of the dependency, generated from a header file.

NOTE : if you inline code in your headers, it will not be possible to mock it! You must separate clearly the declaration (in a header) and the definition (in a C or C++ file).

3.2.2 Let's fizzbuzz again

We will be using the same sample project than for micro testing. But this time, let's imagine that there's a sound machine. This sound machine is a combination of software and hardware. The hardware will play some voice recording like “FIZZ”, “BUZZ”, “FIZZBUZZ”. The hardware comes with a software interface so that you can use it.

Unfortunately, this hardware is very costly, and you don't have one at hand. You'll have to stub its public interface to test your code.

You will find all source code for this example in the `fizzbuzz_pure_mock` folder of the binary distribution.

3.2.3 Mocking the sound machine interface

The sound machine interface is provided. It is very simple as it contains a single function:

```
#ifndef SOUND_H_
#define SOUND_H_
int do_sound(char *sound);
#endif
```

We want to call this function from our fizzbuzz implementation like this:

```
#include "fizzbuzz.h"
#include "sound.h"

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char * fizzbuzz(int i)
{
  char *result = calloc(1, 20);

  if (!(i % 3)) strcpy(result, "FIZZ");
  if (!(i % 5)) strcat(result, "BUZZ");

  if(!strlen(result)) sprintf(result, "%d", i);

  int res = do_sound(result);
  if(res != 0) {
    sprintf(result, "ERROR");
  }
  return result;
}
```

To stub or mock this interface, we need to provide its implementation. We can do this by writing it ourself; or by generating it. Let's use opmock for generation. In the following command line, $SUPPORT should generally be replaced by the path to the support folder that you installed together with opmock.[1]

```
>opmock2 -i sound.h -o . -q \
      -I$SUPPORT/support/gnulibc/include \
      -I$SUPPORT/support/gcc_headers/include
```

Will produce two files in the current folder:

```
sound_stub.h
sound_stub.c
```

These two files usually don't need to be modified. If you need to modify them, that's probably a bug in opmock!

3.2.4 Modifying the build

Once you have generated the mock files, you have to include them in your build. We add as well a simple target to call the mock generation phase. You just need to include the derived object files in your build. For full details, have a look to the sample project **`fizzbuzz_pure_mock`**.

3.2.5 Using mocks in the tests

3.2.5.1 Using ExpectAndReturn

If you try to build now, without modifying the existing tests, the tests will run... Or not. The behavior of some tests may look random.

This is because you've not defined any behavior for your mocks yet. The mock is not expecting any calls, so when called it has to return a default value, which is random. If you're lucky, this value will be the one expected in your assertions, and the test will pass.

When generating C mocks, mocking works at the function level. Let's take the first test as an example:

```
void test_fizzbuzz_with_3()
{
  do_sound_ExpectAndReturn ("FIZZ", 0, cmp_cstr);
  char *res = fizzbuzz(3);
  OP_ASSERT_EQUAL_CSTRING("FIZZ", res);
  free(res);
}
```

Here, we program the behavior of a mock function. When we write

1 There are other options for the include path. Please have a look to the detailed description of command line options to know more.

```
do_sound_ExpectAndReturn("FIZZ", 0, cmp_cstr);
```

It means : *“next time the operation do_sound is called with parameter param of value “FIZZ”, return 0. And please, check that the parameter has the expected value using the comparison function pointed by cmp_cstr”*.

The function signature is the same than the original function, plus some additional parameters:

- The value you want to return when the function is called
- A list of ***matcher*** pointers you use to check input parameters. Matchers are simple C functions that will compare one input value and the expected value when calling a function. Some matchers are provided by opmock for common types. Here, `cmp_cstr` is a matcher function that will compare 2 C strings and return 0 if they're equal. You can use custom matchers : learn more about them in the dedicated section of this document.

You can make several calls to `do_sound_ExpectAndReturn`. The expected parameters, return values, and matchers are stored in an internal FIFO call stack for the function. When you actually call the function, the return value is popped from the stack.

If you call the operation too many times (the call stack is empty), the mock will complain about an unexpected call.

If you call the operation with parameters that don't match the recorded parameters, the mock will complain as well.

If you call functions in a wrong order, opmock will capture it as well.

Please note that, because opmock does not do any memory allocation, the call stack has a limited size, currently set to 100 calls. You can still do more than 100 calls (for example in a loop) if you call `ExpectAndReturn` everytime you're going to call the mocked function, rather than building the full call stack upfront.

Opmock checks all the conditions above. However, it will not fail your test unless you call either the OP_VERIFY or the OP_VERIFY_NO_ORDER macros. These macros will check the recorded errors in the scope of the current test, and fail the test if necessary.

Note : if you use another unit testing framework, you should have a look to the implementation of the OP_VERIFY macros to see how to implement the verify step.*

3.2.5.2 When you don't want to check parameters

Sometimes, when you call a mocked function, you just want the mock to return a value, but you don't care about the parameters values. But you still have to provide matcher pointers when setting the mock behavior...

Say you've mocked the following function:

```
int MyFunction(int i, float j, char *val, MyStruct one_struct);
```

And that when you call the mocked version of the function, you don't want to check the parameters. In this case, just provide NULL matchers:

```
MyFunction_ExpectAndReturn(1, 0.5, “Hello World”, one_struct, -1,
                NULL, NULL, NULL, NULL);
```

By providing a NULL pointer for a matcher, you actually tell Opmock to skip parameter checking when the mock is called.

You can also choose to check some parameters, and ignore others:

```
MyFunction_ExpectAndReturn(1, 0.5, “Hello World”, one_struct, -1,
                cmp_int, NULL, cmp_cstr, NULL);
```

3.2.5.3 Using Callbacks

Mocking with ExpectAndReturn works well when your function has simple parameters and output values. In real code, you'll often have situations where the mock behavior should be more complex.

For example:

```
char *do_something(struct complex_struct * struct_pointer, int *error_code);
```

This operation is supposed to fill in the complex structure pointed by `struct_pointer`, and to fill in the error_code if something bad happens. Obviously, you can't do this with `ExpectAndReturn`, because you can return only simple values.

In this case, you can use a callback function. A callback function is actually a stub : a piece of code you write to mimic the behavior of the actual implementation. Opmock2 has an advantage though over traditional stubbing : your stubs are defined in the scope of your tests, and you can easily swap stub implementations during the test.

Let's try this in the scope of the test `test_fizzbuzz_with_15`.

Your callback function must follow a specific prototype. There's a prototype for each function. This prototype can be found in the generated `sound_stub.h` file:

```
typedef int (* OPMOCK_do_sound_CALLBACK)(char *  sound, int calls);
```

It is exactly the same signature than the function you want to stub, with an additional parameter : `int calls`. Opmock will fill this parameter with the number of times the callback has been called. This can be useful if you want to adapt the behavior of your stub depending on the call sequence.

Let's implement our callback, inside the test file :

```
static int test_fizzbuzz_with_15_callback (char *  sound, int calls)
{
  if((strcmp(sound, "FIZZ") == 0)
     || (strcmp(sound, "BUZZ") == 0)
     || (strcmp(sound, "FIZZBUZZ") == 0))
    return 0;
  return 1;
}
```

And let's use it in out test:

```
void test_fizzbuzz_with_15()
{
  do_sound_MockWithCallback (test_fizzbuzz_with_15_callback);
  char *res = fizzbuzz(15);
  OP_ASSERT_EQUAL_CSTRING("FIZZBUZZ", res);
  free(res);
}
```

Every time we call the `fizzbuzz` function, our callback will be called. You can use a mix of callbacks and `ExpectAndReturn` in your tests.

NOTE : callbacks are not stored on the mock call stack. When you switch to callback mode for a function, you reset the mock stack.You can however mix sections of a test where you use only callbacks, and sections where you use only mocks.

3.2.5.4 Verifying your mock

A function mock records all expected calls to a function. This includes :

- parameters
- return value, if there's one
- call order of operations/functions

After running a test, and after checking your assertions, it can in addition check if your mock was called properly. This is called a “verify” phase.

There's a macro specially for this in the Micro Testing part of Opmock.

Writing:

```
OP_VERIFY();
```

At the end of your test it will be enough to check if **all the mocks for all the functions** were called properly. This macro will exit the test with an error if the verify phase has failed.

If you don't call this macro at the end of your test, the mock errors are still recorded,

but the test will not fail.

An additional macro is available:

```
OP_VERIFY_NO_ORDER();
```

This macro has the same effect than OP_VERIFY, but will not fail the test if the call sequence of mocks is not correct. For example, if your code was supposed to call a function “lock” then a function “unlock”, but has done the opposite, using OP_VERIFY will fail the test, but using OP_VERIFY_NO_ORDER will not.

4 Opmock tutorial for C++

Opmock for C++ follows the same lines than the C version. However, because C++ is much more complex than C, some features of the language are not yet supported:

- You can't mock template classes or template functions. By definition, template classes or functions need to be instantiated before you can use them, and this does not happen in the headers that opmock can parse. Long story made short, if you use templates you can unit test them but you will not mock them – you can still mock their dependencies of course.
- Friend functions are not managed
- Constructors and destructors have default empty bodies. You can't yet call the constructor of the super class. This is not a real issue, as constructors typically have no other side effect than creating an object. You can always populate the object attributes yourself if you need to.
- Operator overload are not mocked properly
- The C++ version is still experimental (but works in most cases). The C version is considered stable for a day to day basis.

4.1 Micro-testing

There are no differences with C : you can use exactly the same macros and functions. This was a design choice not to use C++ features in the unit testing framework.

Generally speaking, you'll use fully scoped operation names instead of function names with C++. For example, mocking a class operation:

```
space1::ClassTest_Mock::func1_ExpectAndReturn(2, 3, 6, cmp_int, cmp_int);
```

Note the suffix **_Mock** at the end of the class name. Every time you will mock a class, a second class in the same name space, with the same name but this suffix, will be generated. This class acts as a delegate of the original class and offers all mocking functions like ExpectAndReturn, Reset, MockWithCallback, and so on.

This way, the contract of the original interface is not modified. Of course, the implementations of the operations of the class are replaced with mocked versions. (excepted inline operations : you should move them in a cpp file if you want to mock the header)

Note as well that all mock operations are static, so that you can mock all instances of

C++ classes, either created on the stack or using new() and even if you can't reach them through a pointer or a reference. This is an advantage over frameworks like Google Mock.

4.2 Using a mock

In C++, what we call a mock is the same than in C. We just use the header file containing the C++ classes we want to mock. We don't link with the genuine implementation. This type of mocking is specially useful at the boundaries of a module or a file.

The sources for this example are in the folder **`cpp_test_no_inheritance.`**

4.2.1 Generating the mock classes

If you have a look to the generate.sh script, you will find the following lines:

```
#!/bin/sh

../../opmock2 -i header.h -o . -cpp -q
-I/home/pascal/dev/opmock/opmock-code/opmock2/support/gnulibc/include \

    -I/home/pascal/dev/opmock/opmock-code/opmock2/support/gcc_headers/include \

    -I/home/pascal/dev/opmock/opmock-code/opmock2/support/gnucpp/include/c+
+/4.2.4 -I/home/pascal/dev/opmock/opmock-code/opmock2/support/gnucpp/include/c+
+/4.2.4/x86_64-linux-gnu \

-x c++ -std=c++98
```

We call the mock generator, using the -cpp yes option because this is a C++ header. Please note the options -x c++ -std=c++98 : these are clang options that can be passed to better specify the C/C++ dialect for this header. Have a look to the end of this document for more informations on command line parameters.

The result of the generation will be 2 files:

```
header_stub.hpp
header_stub.cpp
```

named after the original header file.

4.2.2 Modifying the makefile

We need to make sure that the mocked classes are in the build. Here's the beginning of our makefile:

```
CPPFLAGS=-O0 -ggdb -Wall
OBJECTS = main.o header_stub.o
```

4.2.3 Using mocks in the tests

4.2.3.1 Global Class Mocking

In the main.cpp file, we have some tests.

```
space3::Class3_Mock::multiply_ExpectAndReturn (4, 4, 16);
ToTest class1;
int res = class1.doSomething(4);// doSomething will call multiply()
```

Here, we use a class mock. This means that all instances of this class will use the function mock behavior, when you call the specified operation... Even instances you've not created yourself in the scope of your test. This is useful if you try to add unit testing to an existing build, but you can't easily cut dependencies to other parts of the build. And these parts use the same class than you, in your unit test.

Because this is a static operation, you can only record parameters and return a value. You can't perform a side effect on a specific class instance.

4.2.3.2 ExpectAndReturn, Verify, Callback

All these operations are class operations. They work exactly the same than their C counterparts, and apply to all instances of a class, for a specific operation. The only difference is that you have to prefix the operation name with the qualified class name with the _Mock suffix, for example:

```
space1::Class1_Mock::func1_MockWithCallback(my_callback);
```

4.2.3.3 Class instance mocking

Most of the time, you want to check side effects on a class you test, not on the dependencies of this class. However, your tested code may check the status of the mock after calling it. For example:

```
ClassDep myDep;
myDep.doSomething(5);
if(myDep.attribute == 12)
{
      //do something
}
```

We assume that `ClassDep` has been mocked.

In this situation, the mock must be able to perform a side effect on itself, because the caller is going to check its attributes after calling. Because the side effect can be complex, it has to be done in a specific callback controlled by the tester:

```
static int class3_callback (int  a, int  b, int calls,
                            space3::Class3 * ptr)
{
  ptr->attribute1 = 42;
  return 1234;
}
...
space3::Class3 class3;
space3::Class3_Mock::multiply_MockWithInstanceCallback(class3_callback);
int res2 = class3.multiply(2, 24);
```

We can see that there's an additional parameter to the callback function : a pointer to a Class3 instance. The mock will fill this pointer with its own address. In the callback, we can then use this pointer to perform a side effect on the mocked class.

4.3 Dealing with overloaded C style functions

In C++, you can have overloaded C style global functions, like:

```
int functionOverload (int i, char j);
int functionOverload (int i, char j, void *z);
int functionOverload (int i, char j, float t);
```

Opmock will generate specific callback code to deal with overloaded functions:

```
typedef int (* OPMOCK_functionOverload_CALLBACK)(int  i, char  j, int
calls);
void functionOverload_MockWithCallback(OPMOCK_functionOverload_CALLBACK
callback);
typedef int (* OPMOCK_functionOverload1_CALLBACK)(int  i, char  j, void *
z, int calls);
void functionOverload_MockWithCallback(OPMOCK_functionOverload1_CALLBACK
callback);
```

Note that we have to use different callback typedefs, one per signature. If you want to define a callback for the second overload of the function, your callback should respect the `OPMOCK_functionOverload1_CALLBACK` type definition. (Note the 1 appended. For the 3d overload this would be a 2).

When you want to mock a specific version of the function, you must of course call the proper version of the expectAndReturn functions, for example, if you want to mock the first form of the function:

```
functionOverload_ExpectAndReturn (int  i, char  j, int to_return,
```

```
OPMOCK_MATCHER match_i, OPMOCK_MATCHER match_j);
```

4.4 Mocking template classes

Mocks for template classes or template functions is not supported yet. (but may be in a future release). However, you can mock functions taking templated parameters.

5 Advanced usages

In this section, we provide some informations on more advanced usages of opmock.

5.1 Writing custom matchers

Opmock comes with some pre defined matchers for common types like int or a c string. However, you'll probably want at some point to write your own matchers, for example to compare 2 complex structures or even to compare 2 classes.

A matcher must respect the following prototype:

```
typedef int (* OPMOCK_MATCHER)( void *  a,
                                void * b,
                                const char * name,
                                char * message);
```

This prototype is defined in the header `opmock.h`.

Your matcher will get:

-a void * on the first parameter to compare (which is the **expected** value)

-a void * on the second parameter to compare (which is the **actual** value, that is, the value the matcher will receive at run time in the scope of a test). Note that you'll likely have to cast your parameter to a void *.

-the name of the parameter to compare

-a pointer on a buffer that will be filled with an error description if the 2 input values are different

A matcher must return 0 if the two parameters are equal, and 1 otherwise.

Below is an example of a custom matcher. You can find it in the samples/c_regression folder in the opmock distribution. This matcher compares 2 structs.

```
static int compare_toto_value(  void *val1, void *val2,
                                const char * name, char *buffer)
{
  Toto *toto1 = (Toto *) val1;
```

```
  Toto *toto2 = (Toto *) val2;
  if(toto1->foo != toto2->foo) {
      snprintf(buffer, OP_MATCHER_MESSAGE_LENGTH,
            "parameter '%s.foo' has value '%d', was expecting '%d'",
            name, toto2->foo, toto1->foo);
    return 1;
  }
  if(toto1->boo != toto2->boo) {
      snprintf(buffer, OP_MATCHER_MESSAGE_LENGTH,
            "parameter '%s.boo' has value '%f', was expecting '%f'",
            name, toto2->boo, toto1->boo);
    return 1;
  }
  return 0;
}
```

WARNING : if the parameters to compare in a matcher are pointers (like char *, or a struct pointer), it's a pointer on the pointer that will be stored by opmock. This means that in your matcher, you need to de reference 2 times the value. An example from the matcher cmp_cstring:

```
int cmp_cstr(void *a, void *b, const char * name, char *message)
{
      char * my_a = *((char **)a);
      char * my_b = *((char **)b);

...
```

NOTE : when you return 1 from a custom matcher, opmock will store your error message in the test error stack automatically. If in your test you call OP_VERIFY() (or if you use an equivalent code with another unit testing framework), the test will fail because of this error.

5.2 Mixing mocks and matchers to avoid the use of callbacks

Let's consider this function you want to mock:

```
typedef struct
{
  int foo;
```

```
  float boo;
  int multiply;
} FooStruct;

void callInOut(FooStruct *foo);
```

This function must perform a side effect on the foo parameter : this is both an in and an out parameter.

You can do this easily with a callback. After all, in a callback, you control exactly what you want to do. But callbacks have some disadvantages when compared to mocks:

- callbacks don't maintain a call stack. This means that you can't check the order of the calls for example. Callback calls are never checked in a VERIFY macro.
- using a callback resets the call stack of a function. The consequence is that you can't mix mocks and callbacks for the same function in the scope of a test (unless you know exactly what you're doing – but this is error prone and fragile).
- You can't record expected value of parameters and compare them with the actual values you get in the test. Of course, you can hard code this in the callback logic, but this is cumbersome.

On the other hand, mocks can:

- record and check parameters
- check the number of mock calls and the order of the calls
- optionally use matchers

But they can't perform side effects on parameters, because they don't have the logic for it : how a mock could know how to fill in a struct for example?

There is however a way to couple the advantages of mocks and the advantages of callbacks:

- Create a custom matcher for parameters for which to want a side effect
- Use these custom matchers together with your mocks
- A custom matcher gets:
    - the expected value of a parameter
    - the actual value of a parameter
- It means that a custom matcher can both check your parameters and fill them in!

There's a working example in the samples/c_matchers of the Opmock distribution. Look at the function `custom_matcher` for all the details!

5.3 Failing a test from within a callback function

Inside a callback function, you are “on your own”. This means that:

- You have to check the input parameters yourself (possibly reusing matchers)
- You handle yourself your “call stack” using the number of calls parameters

Mocks can make a test fail when a condition is not met (bad parameters, bad number of calls). You can do the same inside a callback, using a simple function:

```
opmock_add_error_message(char * error_message);
```

When you call this function with an error message, opmock will store the error in the test error stack. At the end of the test, if you call the `OP_VERIFY()` macro, it will fail the test. If you don't use the opmock micro testing framework, you can still retrieve the number of errors in the test using `opmock_get_number_of_errors()` then check the array of error messages.

NOTE : you don't need to call this function from custom matchers. The return value of a matcher is enough for opmock to record automatically an error for the test.

6 Opmock additional tools

Opmock comes currently with a simple helper script to generate test files for you.

6.1 Generating your test suite automatically

If you've chosen to use the unit testing framework that comes with opmock, then every time you add a new test, you need to:

- Modify the C or C++ file where the test implementation is located
- Modify the header file where the test signature is defined
- Modify the main file to register the new test

It is somewhat tedious to maintain all these files, and one can easily forget to register a test. Opmock provides a very simple utility script in the scripts folder:

`refresh_tests.sh.`

This script will:

- Read a list of C and/or cpp test files
- Parse each file to find tests. To find your tests, they must follow a specific signature, which is : `void test_name_of_the_test()`. Note that you can have blank spaces or tab before the void, this will work as well.
- Generate a header file matching the test. If the test file is named “mytest1.c” then the header file will be named “mytest1.h”. You should include this header

in your test file. You can have several test files : a header will be generated for each of them.

- Generate a main.c file registering and running all your tests.

Calling the script is simple:

```
refresh_tests.sh fizzbuzz_test.c fizzbuzz_test2.c
```

Will generate 2 header files and a main.c file. You may have to rename the main.c to main.cpp if you have link time errors with a mix of C and CPP files.

There's a fully working example, including Makefile rules, in the samples/c_matchers folder of the Opmock distribution.

Accepted extensions for C test files are:

- .c, .C, and will result in .h files.

Accepted extensions for C++ test files are:

- .cc; .cpp, .c++, .CC, and will result in .hpp files.

6.1.1 Ignoring a test

You can easily ignore a test during code generation. Just comment it!

You **must** put the comment start on the same line than the test signature:

```
/*void test_afizzbuzz_with_3()
{
  do_sound_ExpectAndReturn ("FIZZ", 0, cmp_cstr);
  char *res = fizzbuzz(3);
  OP_ASSERT_EQUAL_CSTRING("FIZZ", res);
  free(res);

  OP_VERIFY();
}*/
```

So that the test signature does not match the regular expression used by the script.

As a good practice, you may want to add a specific token (like “ignore”) inside your comment, so that you can find quickly ignored tests. Note that having lots of commented tests is a bad code smell...

7 Using Opmock with other unit testing frameworks

The mocking functionality of Opmock is independent of a unit testing framework. You can use Google Test or CppUnit or any other framework if you like and if you need more than the basic functions offered by opmock out of the box. Read this section to discover how to do this.

7.1 How does Opmock proceed

When you use the built-in unit testing framework, all your tests must have the signature:

```
void func1();
void func2();
```

and you must register your tests against opmock:

```
opmock_test_suite_reset();
opmock_register_test(func1, "func1");
opmock_register_test(func2, "func2");
// register another test ...
opmock_test_suite_run();
```

The call to `opmock_test_suite_run()` will execute each test one after the other, in the order they were registered.

Before each test, opmock resets the mocks call stack and states.

After each test, opmock checks the mock errors. However, it does not fail the test, it just run the verify phase.

7.2 Using no testing framework

If you use another unit testing framework, at the beginning of each test you must call explicitly the function:

```
opmock_test_reset();
```

and at the end of each test you may call explicitly:

```
opmock_test_verify();
```

You need to call opmock_test_verify() only if you want to fail the test on mock errors (bad parameters, bad calls).

For example, if you want to test without any testing framework, hence just using

genuine C functions, your test would look like:

```
void my_test
{
      opmock_test_reset();
      // your pre-conditions (input parameters)
      // setup your mocks behavior
      // exercise the code under test
      // check your assertions
      opmock_test_verify();
      int errors = opmock_get_number_of_errors();
      // if you want to check errors excepted call order, use
      // opmock_get_number_of_errors_no_order()
      // instead
      if(errors) {
            opmock_print_error_messages();
            printf ("Verify failed in file %s at line %d\n",
__BASE_FILE__, __LINE__); \
            // do something to deal with the test error
      }
}
```

Beside the `opmock_test_reset()` and `opmock_test_verify()` functions, you have access to two utility functions:

```
opmock_get_number_of_errors();
```

will return you the number of errors in the scope of the current test. This includes errors generated by *matchers*, and errors generated by the verify phase (like bad parameters when calling a mock, or bad number of calls to a mock, or bad call sequence).

```
opmock_get_number_of_errors_no_order();
```

Same than above, but ignore errors related to the call sequence. Calling func1, func2 instead of func2,func1 will not trigger an error.

```
opmock_print_error_messages();
```

will print on stdout all error messages recorder so far. If you're not satisfied by the output of this function, you can access the array where error messages are stored using the variables:

```
extern opmock_test_struct opmock_test_array [OP_MAX_ERROR_MESSAGE];
extern int opmock_error_message;//contains the current number of errors
```

7.3 Using a unit testing framework and fixtures

If you use Google Test, or CppUnit, or any full featured unit testing framework, you'll probably have fixtures at hand – that is, pre and post conditions you can apply to all tests.

In this case, in the pre-condition for a test call `opmock_test_reset()`, and in the post-condition call `opmock_test_verify();`

Use the additional functions to get errors and display them, and to fail the test if you want to.

8 Dealing with Macros

To define macros, just use the standard -D option on the command line. For example, -Dfoo=boo.

9 Known limitations

9.1 Opmock for C

9.1.1 Global variables

Opmock just mocks C functions, but it does not copy the global variables that could be declared (as extern) in a header file. The developer hence will have to add these variables to its build (for example in a test file) to resolve link time dependencies.

9.2 Opmock for C++

9.2.1 Templates

Template classes and template functions are not supported as they should be instantiated before use.

9.2.2 Friend functions

Friend functions are not managed properly.

9.2.3 Operator overloading

This is not currently supported nor tested.

10 Some technical details

10.1 Motivation

There are many existing unit testing frameworks for C and C++, like cppunit, cxxtest, Check, Ctest, Unity, Google Test, and others. There are also some stubbing frameworks for C, like Cmock, and a few for C++. And of course, there are some mocking frameworks, the most advanced being with no doubt Google Mock.

So why create yet another framework ? Actually, if you have been working with large legacy software, you probably faced situations where :

- You can't use fancy frameworks like Google Mock, because you can't use RTTI or even exceptions. Or you may have to stick to a very old compiler version. Or you can't make all your operations virtual.
- You can't use most testing frameworks because the setjmp/longjmp calls are manipulated by the code you're testing.
- You can't even safely allocate memory, because all operators are overloaded and even the bootstrap sequence (gcrt.o for the GNU compiler) of the compiler has been replaced by a different one, to make sure people don't use regular new() or delete() or malloc().
- Your code base is so large and so intricate that your first problem is stubbing rather than testing – writing unit tests is easy. But running a build in an acceptable time to enable TDD is another story when everything depends on everything !
- You don't control the build system – It is large and includes many dependencies that you can't remove, unless you undergo a massive refactoring of the code – something which most of the time is not acceptable. Still, you want to write unit tests, but not break the existing test build.
- You can't easily modify the existing code. For example, you can't introduce virtual functions in classes.

The author has used all these frameworks and met different issues with tricky code bases. Hence it was decided to create another framework, that would work with most if not all situations.

10.2 Advantages

Opmock is available both for C and C++. These two versions have quiet a number of differences.

The C version :

- Does not allocate memory (no use of malloc()), so it should work even when the default implementation of malloc has been replaced. All required structures are allocated in the data section of the executable. Can be seen as well as a disadvantage but it is a design choice. On popular request I could implement also a version using dynamic memory allocation.
- Does not fiddle with signals or setjmp/longjmp.
- Records all calls and can simulate return values. Can check if a calling sequence is correct.
- Can also redirect the calls to a callback function if the behavior to simulate is too complex.

Beside these stubbing and mocking functionalities, Opmock also provides the typical

set of assertion macros found in any unit testing framework, and a very light set of functions to manage test suites. You don't have to use them if you don't want to; it is perfectly possible to use only the mocking part, or the testing part, or to mix them with other frameworks, like Google Test or CppUnit.

The C++ version :

- Does not allocate memory (no use of new() or delete() nor malloc() or free())
- Does not fiddle with signals or setjmp/longjmp
- Records all calls and can simulate return values
- Can also redirect the calls to a callback function if the behavior to simulate is too complex
- Mocks can apply to **all instances** of a class – not only instances you created yourself, and even if they're static or declared on the stack. This is a design choice in opmock, so that you can mock classes even if you don't have direct access to them (through a pointer for example). Again, this happens quiet often when working with legacy code where “dependency injection” is a remote dream.
- Mocks can also apply to a class instance, if you can get a pointer to it. This is mandatory if you want to perform a side effect on the instance, like modifying its attributes.

10.3 Drawbacks

Opmock is a code generator. It can't modify classes at runtime like some frameworks do (Ie Mockito in Java, RhinoMock in C#, Google Mock in C++).

Opmock does not cover currently the full C++ syntax. There are situations where you'll have to fix the generated files manually. Still, this should save you a **lot** of time.

Because Opmock does not do any memory allocations, there are some limitations on the number of calls you can register against a mock. This can be fixed by changing a single line in the generated code, or using mock Reset functionalities.

Because Opmock is based on code generation, it can't recognize automatically parameters types (RTTI would allow that).

10.4 Is Opmock the right tool for you ?

It depends. If you work on a modern C++ project and a clean code base, with decent compilers (say gcc 4.x or Visual C++) then you may find that Google Mock and Google Test are better options, for C++ at least. However, Google Mock still requires that you link with the symbols you want to mock. In other words, you have to link at least with a stub before you can mock at runtime... Opmock generates both stubs and mocks in one go.

You could have technical impediments though, if you work with C code or a C++ product playing hard with signals, memory allocation, operator overloading, and bootstrapping sequences of the executables (and specially the global constructor init phase). The way you build your product can have as well a big impact. It's not always possible to break a build in small, manageable pieces when you work on legacy code. So you end up embedding in your test build tons of code that you don't use directly.

Opmock can cope with these extreme but not so uncommon situations. While it's based on a stubbing strategy, it will generate most of the code for you, making the journey to TDD a much smoother one.

If you work with pure C code or a mix of C and C++, then Opmock may be one of your best options.

11 A few words on stubbing and mocking

11.1 Stubbing, mocking, both?

In this document, the *stubbing* and *mocking* words are frequently used. You can find an academic definition of those from Martin Fowler (http://martinfowler.com/articles/mocksArentStubs.html).

Opmock actually generates stubs, that is, minimal implementations of an interface. But at the same time, you can program the behavior of this stub and verify the call sequence : this is typical of a mock. Opmock hence fits both definitions.

My personal style (that you don't have to follow if you have yours) when testing is:

- Set the input parameters
- Optionally setup some mocks but use NULL matchers
- Call the System Under test
- Check the result (use assertions)

By default, I try to avoid the use of mocks, because they're tightly coupled to internal implementation details of the SUT. If my test can be considered as “black box testing” because I just use the public interface of the SUT, using a mock involves some form of “white box testing”, where you know the details of the implementation.

There are situations however, where you do want to check if the dependency was called properly (for example, exactly twice with a given set of parameters). I think that these situations are not very common though, unless your code is a kind of state machine.

However, Opmock supports this also. Our test flow becomes:

- Set the input parameters
- Set the mock expectations (use expect and return)
- Call the System Under test
- Check the result (use assertions)
- Check mock expectations (use verify)

Alternatively, you may safely skip the last phase. In this case, Opmock still records the parameters and output values, but it is transparent for you. This is a particular style of stubbing.

11.2 Fixtures with Opmock

You may have realized that there are no setup() or teardown() operations in Opmock. This is a design decision. Too often have I seen these operations used to trigger a behavior so complex that:

- the tests were not micro tests anymore, but rather functional tests,
- the tests could not be executed in any order (think isolation),
- the tests could not be executed twice unless you clean up an external system first (like a database).

Opmock can still support this, because when you register tests (or if you run them manually), the tests are executed in the order they were registered. If really you want a setup and a tear down, register them as the first and the last test, respectively. Or use utility operations that you can call in each test.

But I strongly advise you against this style. Your micro tests should always be executable in any order, any number of times.

12 Command line options

This section gives the extensive list of all opmock options, as well as some useful SWIG options.

12.1 -i

Use this option to specify the input header file from which you want to generate code:

```
-i ../../myfile.h
```

The file name can either use a relative path, or an absolute path. If your path contains white spaces, you may want to put double quotes “” at the beginning and at the end of the path.

12.2 -o

Use this option to specify the output path where code is going to be generated. The output file names (header file and C/C++ file) will always follow the input file name. For example, if your input file is named `header.h`, the generated files will be `header_stub.h` and `header_stub.c`.

```
-o .
```

The path can either be a relative path, or an absolute path. If your path contains white spaces, you may want to put double quotes “” at the beginning and at the end of the path.

12.3 -cpp

Use this option to tell Opmock if you to generate code from a C or rather a C++ header file.

```
-cpp
```

to generate C++ code.

Just don't use this option to generate C code.

12.4 -s

Use the option to specify a list of functions or class operations for which *you don't want* to generate code (s for *skip*). This can be handy if you just want a subset of functions, or if you have weird header files declaring functions as extern.

The list of functions to exclude is given as a comma separated list of names. Beware that in C++, because you can have operation overloads, specifying an operation (like ::MyClass::MyFunction) will disable code generation for all overloads of this function.

```
-s function1,function2,function3
```

can be used to skip C function generation

```
-s ::Class1::operation1,::Class2::operation2
```

can be used to skip the corresponding operations.

12.5 -k

This option is the opposite of the option “-s”. Use it to specify a list of functions or operations that you *want to keep* in the generated code. All other functions will be ignored. This option is handy if you just want to generate a few functions from a large header file.

--keep-only `function1,function2,function3`

for C, or:

```
--keep-only ::Class1::operation1,::Class2::operation2
```

for C++.

As for --skip-funct, all overloads of a class operation are kept.

12.6 -q

Use the option -q for opmock to include the original header file with quotes.

-q

will generate code like:

#include "dep.h"

On the other hand, if you *don't use* this option, the generated code will be:

#include <dep.h>

12.7 -p

Use the option -p to specify a prefix when including the original header file.

-p myprefix1/myprefix2

will generate code like:

#include <myprefix1/myprefix2/dep.h>

This option is useful if you need a specific prefix to include the original header file. This can happen in complex build systems.

12.8 -I

The -I option is a compiler include option : it will be passed verbatim to clang for headers and files inclusions.

Opmock always starts with an **empty** include path : you need to provide at least a basic include path to find standard C headers (stdio.h for example) and/or C++ headers (standard lib or stl). And you need to add any extra paths used by your code.

Opmock comes with some headers extracted from gcc 4.2.x. You can use those instead of you system headers for standard lib c and c++ standard lib and STL. This is not mandatory : you can also use the headers provided with your compiler, but you may have some issues. For example, Clang (on top of which opmock is based) does not understand properly some visual C++ headers because of non standard extensions.

Please have a look to the examples in the *samples* subfolders for working examples.

12.9 -x C++ -std=c++98

You can pass to opmock additional options that are, in fact, Clang options. This way, you can further specify the C or C++ dialect used by the header.

Please note that headers ending with .h will by default be parsed as C headers. In this case, make sure you use the -x C++ option so that it is parsed as C++.

12.10 Other options

Opmock can accept all Clang 3.2 options : they will be passed verbatim to Clang. For more informations, please refer to the Clang manual.