Cpputest

CppUTest unit testing and mocking framework for C/C++

CppUTest coverage 100%

- Core Manual
- CppUMock Manual
- Plugin Manual
- Platforms stories
- View on GitHub

Download Release 3.8 as .zip Download Release 3.8 as .tar.gz

CppUTest has support for building mocks. This document describes the mocking support. A couple of design goals for the mocking support were:

- Same design goals as CppuTest – limited C++ set to make it well suitable for embedded soft.
- No code generation
- No or very few magic hiding macros
- Very simple to use
- The developer stays in control

The main idea is to make manual mocking easier, rather than to make automated mocks. If manual mocking is easier, then it could also be automated in the future, but that isn't a goal by itself.

Table of Content

Simple Scenario

About the simplest scenario is to check that one particular function call has happened. The below code is an example of this:

```
#include "CppUTest/TestHarness.h"
#include "CppUTestExt/MockSupport.h"

TEST_GROUP(MockDocumentation)
{
    void teardown()
    {
        mock().clear();
    }
};
```

```
void productionCode()
{
    mock().actualCall("productionCode");
}

TEST(MockDocumentation, SimpleScenario)
{
    mock().expectOneCall("productionCode");
    productionCode();
    mock().checkExpectations();
}
```

The only additional include for mocking is CppUTestExt/MockSupport.h which is the main include for anything CppUTest Mocking. The declaration of the TEST_GROUP is the same as always, there is no difference.

The TEST(MockDocumentation, SimpleScenario) contains the recording of the expectations as:

```
mock().expectOneCall("productionCode");
```

The call to mock() returns the global MockSupport (more about that later) on which we can record our expectations. In this example, we'll call expectOneCall("productionCode") which… records an expectation for *one* call to a function called productionCode.

The productionCode call is to show a call to whatever your real code is. The test ends with checking whether the expectations have been met. This is done with the:

```
mock().checkExpectations();
```

This call is needed when *not* using the MockSupportPlugin, otherwise this is done automatically for every single test. Also, the call to mock().clear() in the teardown is *not* needed when using the MockSupportPlugin, otherwise it is needed to clear the MockSupport. Without the clear, the memory leak detector will report the mock calls as leaks.

The actual mocked function call looks like:

```
void productionCode()
{
    mock().actualCall("productionCode");
}
```

where we use MockSupport by calling mock() and then record the actual call to the productionCode function.

This scenario is quite common when using linker stubbing of e.g. a 3rd party library.

If the call to productionCode wouldn't happen, then the test would fail with the following error message:

```
ApplicationLib/MockDocumentationTest.cpp:41: error: Failure in TEST(MockDocumentation, SimpleScenario)
    Mock Failure: Expected call did not happen.
    EXPECTED calls that did NOT happen:
        productionCode -> no parameters
    ACTUAL calls that did happen:
        <none>
```

Sometimes you expect *several identical calls* to the same function, for example five calls to productionCode. There is a convenient shorthand for that situation:

```
mock().expectNCalls(5, "productionCode");
```

Using Objects

Simple scenario using objects There is no difference between mocking functions and objects. Below code shows a mock using run-time mocking:

```
class ClassFromProductionCodeMock : public ClassFromProductionCode
{
public:
    virtual void importantFunction()
    {
        mock().actualCall("importantFunction");
    }
};

TEST(MockDocumentation, SimpleScenarioObject)
```

```
{
    mock().expectOneCall("importantFunction");

    ClassFromProductionCode* object = new ClassFromProductionCodeMock; /* create mock instead of real thing */
    object->importantFunction();
    mock().checkExpectations();

    delete object;
}
```

The code is self-explanatory. The real object is replaced by a hand-made mock object. The call to the mock then records the actual call via the MockSupport.

When using objects, we can also check whether the call was done on the right object, via this:

```
mock().expectOneCall("importantFunction").onObject(object);
```

and the actual call would then be:

```
mock().actualCall("importantFunction").onObject(this);
```

If the call to a wrong object happens, it would give the following error message:

```
MockFailure: Function called on a unexpected object: importantFunction
    Actual object for call has address: <0x1001003e8>
    EXPECTED calls that DID NOT happen related to function: importantFunction
        (object address: 0x1001003e0)::importantFunction -> no parameters
    ACTUAL calls that DID happen related to function: importantFunction
        <none>
```

Parameters

Of course, just checking whether a function is called is not particularly useful when we cannot check the parameters. Recording parameters on a function is done like this:

```
mock().expectOneCall("function").onObject(object).withParameter("p1", 2).withParameter("p2", "hah");
```

And the actual call is like:

```
mock().actualCall("function").onObject(this).withParameter("p1", p1).withParameter("p2", p2);
```

If a parameter isn't passed, it will give the following error:

```
Mock Failure: Expected parameter for function "function" did not happen.
    EXPECTED calls that DID NOT happen related to function: function
        (object address: 0x1)::function -> int p1: <2>, char* p2: <hah>
    ACTUAL calls that DID happen related to function: function
        <none>
    MISSING parameters that didn't happen:
        int p1, char* p2
```

Objects as Parameters

withParameters can only use int, double, const char* or void* . However, parameters are often objects of other types and not of the basic types. How to handle objects as parameters? Below is an example:

```
mock().expectOneCall("function").withParameterOfType("myType", "parameterName", object);
```

When using withParameterOfType, the mocking framework needs to know how to compare the type and therefore a Comparator has to be installed before using parameters of this type. This is done using installComparator, as below:

```
MyTypeComparator comparator;
mock().installComparator("myType", comparator);
```

MyTypeComparator is a custom comparator, which implements the MockNamedValueComparator interface. For example:

```
class MyTypeComparator : public MockNamedValueComparator
{
public:
    virtual bool isEqual(const void* object1, const void* object2)
    {
        return object1 == object2;
```

```
    }
    virtual SimpleString valueToString(const void* object)
    {
        return StringFrom(object);
    }
};
```

The isEqual is called to compare the two parameters. The valueToString is called when an error message is printed and it needs to print the actual and expected values. If you want to use normal C functions, you can use the MockFunctionComparator which accepts pointers to functions in the constructor.

To remove the comparators, all you needs to do is call removeAllComparatorsAndCopiers, like:

```
mock().removeAllComparatorsAndCopiers();
```

Comparators sometimes lead to surprises, so a couple of warnings on its usage:

Warning 1:

- Pay attention to the scope of your comparator variable!

Comparators are *not* copied, instead it uses the exact instance as passed to the installComparator function. So make sure it is still in-scope when the framework tries to use it! For example, if you installComparator inside the TEST, but do the checkExpectations in the teardown, then it is likely to cause a crash since the comparator has been destroyed.

Warning 2:

- Pay extra attention to scope when using the MockPlugin

When using the MockPlugin (recommended), then it's best to install the comparators via the MockPlugin or put them in global space. The checkExpectations will be called *after* teardown and if your comparator was destroyed in the teardown then this will cause a crash.

Output Parameters

Some parameters do not represent data passed to the called function, but are passed by reference so that the function can 'return' a value by modifying the pointed-to data.

CppUMock allows the value of these output parameters to be specified in the expected call:

```
int outputValue = 4;
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &outputValue, sizeof(outputValue));
```

…and written during the actual call:

```
void Foo(int *bar)
{
    mock().actualCall("Foo").withOutputParameter("bar", bar);
}
```

After the actual call, the bar parameter passed to function Foo will have the value specified in the expected call (4, in this case).

Warning 1:

- CppUMock *does not* and *cannot* prevent invalid memory accesses when using output parameters. It will memcpy exactly the number of bytes specified in the withOutputParameterReturning call. A segmentation fault may occur if this is larger than the data pointed to by the output parameter provided in the actual call.

Function overloads of withOutputParameterReturning are provided for char, int, unsigned, long, unsigned long, and double types so that the size parameter may be omitted:

```
char charOutputValue = 'a';
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &charOutputValue);

int intOutputValue = 4;
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &intOutputValue);

unsigned unsignedOutputValue = 4;
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &unsignedOutputValue);
```

```
long longOutputValue = 4;
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &longOutputValue);

unsigned long unsignedLongOutputValue = 4;
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &unsignedLongOutputValue);

double doubleOutputValue = 4;
mock().expectOneCall("Foo").withOutputParameterReturning("bar", &doubleOutputValue);
```

Warning 2:

- When a char, int, etc. array is passed to withOutputParameter, you must use the generic withOutputParameterReturning and provide the actual size of the array or only one element will be copied.

Output Parameters Using Objects

By far the best way to handle output parameters is by using a custom type copier (v3.8). The general principle is similar to the custom comparators described above:

```
MyType outputValue = 4;
mock().expectOneCall("Foo").withOutputParameterOfTypeReturning("MyType", "bar",  &outputValue);
```

The corresponding actual call is:

```
void Foo(int *bar)
{
    mock().actualCall("Foo").withOutputParameterOfType("MyType", "bar",  bar);
}
```

When using withOutputParameterOfTypeReturning, the mocking framework needs to know how to copy the type and therefore a Copier has to be installed before using parameters of this type. This is done using installCopier, as below:

```
MyTypeCopier copier;
mock().installCopier("myType", copier);
```

MyTypeCopier is a custom copier, which implements the MockNamedValueCopier interface. For example:

```
class MyTypeCopier : public MockNamedValueCopier
{
public:
    virtual void copy(void* out, const void* in)
    {
        *(MyType*)out = *(const MyType*)in;
    }
};
```

To remove the copier, you need to call removeAllComparatorsAndCopiers, like:

```
mock().removeAllComparatorsAndCopiers();
```

Warning 1 and *Warning 2* above apply to copiers as well.

Return Values

Sometimes it is needed to let a mock function return a value which can then be used in production code. The test code would look like this:

```
mock().expectOneCall("function").andReturnValue(10);
```

The mock function would look like:

```
int function () {
    return mock().actualCall("function").returnIntValue();
}
```

or we could separate returnIntValue from the actualCall (below it!) like:

```
int function () {
    mock().actualCall("function");
    return mock().intReturnValue();
}
```

The return value options are used to transfer data between the test and the mock object, they themselves do not cause the tests to fail.

Passing other data

Sometimes a test wants to pass more data to the mock object to, for example, vary only a couple of parameters in a calculation. This can be done like this:

```
ClassFromProductionCode object;
mock().setData("importantValue", 10);
mock().setDataObject("importantObject", "ClassFromProductionCode", &object);
```

And it can be used in the mock object like:

```
ClassFromProductionCode * pobject;
int value = mock().getData("importantValue").getIntValue();
pobject = (ClassFromProductionCode*) mock().getData("importantObject").getObjectPointer();
```

Like return values, setting data will not ever make a test fail but it provides support in building mock objects.

Other MockSupport - ignoring, enabling, clearing, crashing

MockSupport offers a couple of other useful functions, which will be covered in this section.

Frequently, you only want to check a couple of calls in your test and ignore all the other calls. If you add expectOneCall for each of these calls, your tests might become too large (though, it might be a smell that your test is indeed too large). One way to prevent this is the ignoreOtherCalls, like:

```
mock().expectOneCall("foo");
mock().ignoreOtherCalls();
```

This will check that one call of foo happens (and only one call!), but all other calls will be ignored (such as “bar”).

Sometimes, you don’t want to just ignore calls, but instead disable the whole mocking framework for a while (to do *something*). This happens sometimes in initialization where you might want to do *something* without the mocking framework checking calls. You can do this by enabling/disabling such as:

```
mock().disable();
doSomethingThatWouldOtherwiseBlowUpTheMockingFramework();
mock().enable();
```

When you are ignoring calls to mocked functions that have a return value there is a catch. You will probably get a runtime error on your test when you try to ignore it.

As almost all things in life it is easier to explain this with an example, lets say you have this mock function:

```
int function () {
    return mock().actualCall("function").returnIntValue();
}
```

If you try to ignore it or disable the framework, it will explode. Why ? The return value is not defined so there is no way to define the return value of the mocked function.

To cases like these there is a series of return value functions that allows you to define a default return value that will be returned when the mock function is ignored.

The example above could be written as:

```
int function () {
    return mock().actualCall("function").returnIntValueOrDefault(5);
}
```

If the function is ignored or the framework is disabled, a 5 will be returned, otherwise the expected return value will be returned (according to the expectations set on the test case).

You can also use mock support directly to do this (instead of using the actual call object):

```
int function () {
    mock().actualCall("function");
    return mock().returnIntValueOrDefault(5);
}
```

If you want to clear all the expectations, settings, and comparators, call clear:

```
mock().clear();
```

Clear won't do checkExpectations, but just erase everything and start over. Usually clear() is called after a checkExpectations.

Sometimes, a mock actual call happens, but you cannot figure out from where it is called. If you only had a call stack, then it you could track it. Well, unfortunately, the mocking framework doesn't print stack traces, but it can crash! If you call the crashOnFailure on the MockSupport, then it will crash so that you can use the debugger to get a stack trace. like:

```
mock().crashOnFailure();
```

When using gdb, get a stack trace using:

```
gdb examples/CppUTestExamples_tests
r
bt
```

(r is run, it will run until crashes. bt is back trace which will produce a stack)

MockSupport Scope

MockSupport can be used hierarchically using MockSupport scope. This sounds really complex, but in reality it is very simple. When getting a mock support using the mock function, you can pass a namespace or scope and record the expectations (or do other things) inside this scope. For example:

```
mock("xmlparser").expectOneCall("open");
```

The actual call then has to look like this:

```
mock("xmlparser").actualCall("open");
```

A call on another namespace won't work, for example this won't match the call to xmlparser open:

```
mock("").actualCall("open");
```

Keeping calls in namespaces makes it easy to ignore one type of call and focus on another, for example:

```
mock("xmlparser").expectOneCall("open");
mock("filesystem").ignoreOtherCalls();
```

MockSupportPlugin

There is a MockSupportPlugin to make the work with mocks easier. Complete Documentation for MockSupportPlugin can be found on the Plugin Manual page.

C Interface

Sometimes it is useful to access the mocking framework from a .c file rather than a .cpp file. For example, perhaps, for some reason, the stubs are implemented in a .c file rather than a .cpp file. Instead of changing over all to .cpp, it would be easier if the mocking framework can be called via C. The C interface is exactly meant for this. The interface is based on the C++ one, so below is some code and it ought to be easy to figure out what it does (if you've read all that was written earlier):

```
#include "CppUTestExt/MockSupport_c.h"

mock_c()->expectOneCall("foo")->withIntParameters("integer", 10)->andReturnDoubleValue(1.11);
return mock_c()->actualCall("foo")->withIntParameters("integer", 10)->returnValue().value.doubleValue;

mock_c()->installComparator("type", equalMethod, toStringMethod);
mock_scope_c("scope")->expectOneCall("bar")->withParameterOfType("type", "name", object);
mock_scope_c("scope")->actualCall("bar")->withParameterOfType("type", "name", object);
mock_c()->removeAllComparators();

mock_c()->setIntData("important", 10);

mock_c()->checkExpectations();
mock_c()->clear();
```

The C interface uses a similar builder structure as the C++ interface. It is far less common in C, but it works the same.

It is now also possible to specify the actual return value in the same way as with C++ (v3.8):

```
return mock_c()->actualCall("foo")->withIntParameters("integer", 10)->doubleReturnValue();
return mock_c()->doubleReturnValue();
```

and to specify a default return value, in case mocking is currently disabled when the actual call occurs (v3.8):

```
return mock_c()->actualCall("foo")->withIntParameters("integer", 10)->returnDoubleValueOrDefault(2.25);
return mock_c()->returnDoubleValueOrDefault(2.25);
```

Miscellaneous

If you want your test to be more explicit about that a certain mocked function call should not occur, you can write (v3.8):

```
mock().expectNoCall("productionCode");
```

Doing so is functionally equivalent to stating no expectations at all.