MicroMock\_C

Contents

[Beginning Mocks 2](#_Toc394490932)

[Supported Types 2](#_Toc394490933)

[Unsupported Types 2](#_Toc394490934)

[Clone 2](#_Toc394490935)

[Comparison 2](#_Toc394490936)

[To String 3](#_Toc394490937)

[Binding to Unsupported Types 3](#_Toc394490938)

[Mock Functions 3](#_Toc394490939)

[MOCK\_FUNCTION 3](#_Toc394490940)

[MOCK\_FUNCTION\_END 3](#_Toc394490941)

[Actual vs Expected Call Comparison 4](#_Toc394490942)

[COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(comparisonType) 4](#_Toc394490943)

[Comparison Types 4](#_Toc394490944)

[Specifying Expected Calls 4](#_Toc394490945)

[Ignoring Function Arguments 5](#_Toc394490946)

[Resetting Expected and Actual Calls 5](#_Toc394490947)

[A Simple Test 5](#_Toc394490948)

# Beginning Mocks

In order to start declaring any mock functions, BEGIN\_MOCK() must be used. Place this macro before declaring mock functions or making any call comparisons. The BEGIN\_MOCK() macro may take at most 1 argument which can be used to store and reference any expected call.

BEGIN\_MOCKS(MyMocks)

…

MyMocks m = STRICT\_EXPECTED\_CALL(hi("Bill", 5));

Following the BEGIN\_MOCK() include the following 3 macros: COMPARISONS(), TO\_STRINGS(), and CLONES(). These 3 macros tell MicroMock\_C how to find and bind any additional comparison, to string, or clone functions. Even if there are no additional comparison, to string, or clone function, these macros are still necessary.

For symmetry, you may use the END\_MOCK macro after creating all of the mocked functions.

## Supported Types

Any of the following types may be used as parameters within a mock function definition without needing to write additional comparison, to string, or clone functions:

* int
* char
* short
* long
* size\_t
* float
* double
* char\*
* void\*
* unsigned long

## Unsupported Types

In order to use any type other than those listed above, appropriate comparison, to string, and clone functions must be written.

## Clone

A clone function for an unsupported type should take 1 argument which is the same type as the unsupported type for which this function is being written and it should return a pointer to a type that will be most helpful when writing the comparison and to string functions. If we want to clone an int\* for example:

int\* int\_ptr\_Clone(const int\* val)

The clone function must allocate memory (by making a call to malloc())for this pointer type, copy the value of the argument into this newly allocated pointer, and return the pointer.

The clone function will determine how the custom type will be represented all throughout the lifetime of that type in MicroMock\_C. MicroMock\_C guarantees that it will only use these cloned types within the comparisons and to string functions. Therefore, it is important that whenever using a custom type, that the void\* arguments are casted and dereferenced properly inside of the comparison and to string functions depending on how they were cloned.

It is important to note that the function does not need to take care of freeing any memory, MicroMock\_C will take care of cleanup.

## Comparison

A comparison function for an unsupported type should return an int and take 2 void\* arguments that must be casted and compared:

int comparison\_function\_name(const void\* a, const void\* b)

Cast the void\* arguments appropriately depending on how the original arguments were cloned in the corresponding clone function.

## To String

A to string function for an unsupported type should return void and take 4 arguments which are a char\* representing the string onto which this value will be concatenated, a size\_t representing the buffer size of the 1st argument, and a void\* representing the argument that should be represented as a string:

void int\_ptr\_to\_string(char\* string, size\_t bufferSize, const void\* arg)

This function must create a string representation of arg and put it into the string argument. The void\* arg must be casted and dereferenced properly depending on how this custom type was originally cloned in the Clone function.

For safety, use the \_s versions of all function inside of string.h.

## Binding to Unsupported Types

After writing any additional clone, comparison, or to string functions for unsupported types they must be bound to those types so that MicroMock\_C knows which functions to call upon encountering a particular unsupported type. As mentioned earlier, the COMPARISONS(), TO\_STRINGS(), and CLONES() macros tell MicroMock\_C how to find and bind any additional comparison, to string, or clone functions.

To bind these, specify the type followed by the function name as arguments to these three macros:

COMPARISONS(int\*, int\_ptr\_compare, ...)

TO\_STRINGS(int\*, int\_ptr\_to\_string, ...)

CLONES(int\*, int\_ptr\_Clone, ...)

These three macros should only be used exactly once. If there are multiple custom types and functions, continue specifying these bindings in the same fashion until all are bound.

See the example below:

BEGIN\_MOCKS(MyMocks)

typedef struct {

int xorigin, yorigin;

int xdistance, ydistance;

} Line;

int line\_compare(const void\* a, const void\* b)

{

return ((Line\*)a)->xdistance != ((Line\*)b)->xdistance || ((Line\*)a)->ydistance != ((Line\*)b)->ydistance;

}

void line\_to\_string(char\* string, size\_t bufferSize, const void\* arg)

{

Line\* line = (Line\*)arg;

sprintf\_s(string, bufferSize, "{%d, %d, %d, %d}", line->xorigin, line->yorigin, line->xdistance, line->ydistance);

}

Line\* line\_Clone(const Line line)

{

Line\* newline = malloc(sizeof(Line));

(\*newline).xorigin = line.xorigin;

(\*newline).yorigin = line.yorigin;

(\*newline).xdistance = line.xdistance;

(\*newline).ydistance = line.ydistance;

return newline;

}

//Methods for my int pointer type

int int\_ptr\_compare(const void\* a, const void\* b)

{

return \*(int\*)a != \*(int\*)b;

}

void int\_ptr\_to\_string(char\* string, size\_t bufferSize, const void\* arg)

{

sprintf\_s(string, bufferSize, "%d", \*(int\*)arg);

}

int\* int\_ptr\_Clone(const int\* val)

{

int\* newValue = malloc(sizeof(int));

\*newValue = \*val;

return newValue;

}

//The following 3 macros are necessary. Only include parameters if you are using your own comparison, to string, or cloning functions.

//Parameters should be ordered with the type followed by the function to which it is bound for the specified operation (comparison, stringify, cloning)

COMPARISONS(int\*, int\_ptr\_compare, Line, line\_compare)

TO\_STRINGS(int\*, int\_ptr\_to\_string, Line, line\_to\_string)

CLONES(int\*, int\_ptr\_Clone, Line, line\_Clone)

/\*Mocks Section\*/

MOCK\_FUNCTION(, int\_ptr\_function, int\*, argument)

printf("My int is: %d\n", \*argument);

MOCK\_FUNCTION\_END(void, )

MOCK\_FUNCTION(, print\_line, Line, line)

printf("Line begins at (%d, %d) with an x distance of %d and a y distance of %d\n", line.xorigin, line.yorigin, line.xdistance, line.ydistance);

MOCK\_FUNCTION\_END(void, )

/\*End Mocks Section\*/

END\_MOCKS

void int\_foo(void)

{

int x = 100;

int\* myIntPointer = &x;

int\_ptr\_function(myIntPointer);

}

void line\_function(void)

{

Line x = { 0, 0, 10, 10 };

print\_line(x);

}

CTEST\_BEGIN\_TEST\_SUITE(Line\_Test\_Suite)

CTEST\_FUNCTION(LINE\_TEST\_PASSES)

{

//Arrange

Line y = { 0, 1, 10, 10 };

STRICT\_EXPECTED\_CALL(print\_line(y));

//Act

line\_function();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS;

}

CTEST\_FUNCTION(LINE\_TEST\_FAILS)

{

//Arrange

Line y = { 0, 1, 9, 10 };

STRICT\_EXPECTED\_CALL(print\_line(y));

//Act

line\_function();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS;

}

CTEST\_END\_TEST\_SUITE(Line\_Test\_Suite)

# Mock Functions

Mock functions should start with the MOCK\_FUNCTION macro which can take any number of arguments for the mock function.

## MOCK\_FUNCTION

MOCK\_FUNCTION(returnType, name, arg1Type, arg1Value, arg2Type, …)

This macro will declare a mock function with the specified name, arguments, and return type. Any code written between MOCK\_FUNCTION and MOCK\_FUNCTION\_END will be included in the function body.

This macro also enables expected and actual call comparison to occur with every function of this name.

For a function with 0 arguments, simply leave out all arguments from the MOCK\_FUNCTION macro and only include a return type and name.

## MOCK\_FUNCTION\_END

MOCK\_FUNCTION\_END(returnType, value)

This macro must end every mock function. The value parameter is the default return value for this mock function. If the mocked function does not return a value, the value parameter is ignored.

BEGIN\_MOCKS(MyMocks)

MOCK\_FUNCTION(void, hello)

printf("Hello!\n");

MOCK\_FUNCTION\_END(void, NULL)

MOCK\_FUNCTION(void, hi, char\*, name)

printf("Hello, %s!\n", name);

MOCK\_FUNCTION\_END(void, NULL)

END\_MOCK

# Actual vs Expected Call Comparison

## COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(comparisonType)

Calling COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(comparisonType) will use a CTest assert to test for equality on the expected and actual calls. Excluding a comparison type or simply using COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS will use the default comparison type, OrderedCompairon.

If particular function call is expected but no matching actual function call occurred, COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS shall fail and respond with an indication of the actual function call (if one exists) and the expected function call.

If a particular mock function was called but was not expected, the CTest assert from COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS shall respond with an indication of the unexpected function.

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS will fail as soon as the first unexpected or missing call occurs and no further calls will be compared.

## Comparison Types

There are currently two supported Comparison Types: OrderedComparison, and UnorderedComparison.

A call to COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS with the OrderedComparison option will ensure that the actual calls appear in the same order as the calls were specified to the EXPECTED\_CALL macro.

A call to COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS with the UnorderedComparison option will only ensure that the every actual function call was expected regardless of the order in which they were called.

## Specifying Expected Calls

Before using the COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS macro, it is important to specify to MicroMock\_C which calls to expect.

The STRICT\_EXPECTED\_CALL(function) macro tells MicroMock\_C to include this function call when comparing expected and actual calls and to compare all of its arguments.

The EXPECTED\_CALL(function) macro tells MicroMock\_C to include this function call when comparing expected and actual calls but to ignore every arguments during the comparison.

## Ignoring Function Arguments

Use either the IGNORE\_ARGS(expected\_call, …) or IGNORE\_ALL(expected\_call) macro to ignore any or all arguments of a particular expected call.

CTEST\_FUNCTION(FOO\_TEST\_ORDERED)

{

IGNORE\_ARGS(STRICT\_EXPECTED\_CALL(manyArgs("Andy", 2, 3.0, 3.14159)), 1);

IGNORE\_ALL(STRICT\_EXPECTED\_CALL(charArg('f')));

MyMocks m = STRICT\_EXPECTED\_CALL(hi("Bill", 5));

IGNORE\_ALL(m);

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

The IGNORE\_ARGS macro expects arguments to be specified not by index but rather by which argument you want to ignore.

Micromock\_C will ignore a request for an argument to be ignored that is out of range.

## Injecting Return Values

The return value for any mocked function may be changed at any time. Use the SET\_RETURN(expected\_call, return\_value) macro to specify the new return value for the specified function.

When the mocked function is called, this new value will be returned rather than the previously specified default return value.

int foo(void)

{

manyArgs("Andy", 2, 3.0, 3.14159);

charArg('f');

return hi("Bill", 5);

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_CALLS\_OUT\_OF\_ORDER\_AND\_UNORDERED\_COMPARISON\_PASSES)

{

//Arrange

STRICT\_EXPECTED\_CALL(charArg('f'));

STRICT\_EXPECTED\_CALL(manyArgs("Andy", 2, 3.0, 3.14159));

MyMocks m = STRICT\_EXPECTED\_CALL(hi("Bill", 5));

SET\_RETURN(m, 100);

//Act

CTEST\_ASSERT\_ARE\_EQUAL(foo(), 100);

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(UnorderedComparison);

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

## Resetting Expected and Actual Calls

Often times it will be necessary to reset the list of expected and actual calls that have been recorded. To do this, use the RESET\_EXPECTED\_AND\_ACTUAL\_CALLS macro. You MUST reset the expected and actual calls in order to clean up after completing a test and moving onto a new one.

# A Simple Test

#include <CTest.h>

#include <MicroMock\_C.h>

BEGIN\_MOCKS

//Methods for my int pointer type

int int\_ptr\_compare(const void\* a, const void\* b)

{

return \*(int\*)a != \*(int\*)b;

}

void int\_ptr\_to\_string(char\* string, size\_t bufferSize, const void\* arg)

{

sprintf\_s(string, bufferSize, "%d", \*(int\*)arg);

}

int\* int\_ptr\_Clone(const int\* val)

{

int\* newValue = malloc(sizeof(int));

\*newValue = \*val;

return newValue;

}

//The following 3 macros are necessary. Only include parameters if you are using your own //comparison, to string, or cloning functions. Parameters should be ordered with the type //followed by the function to which it is bound for the specified operation (comparison, //stringify, cloning)

COMPARISONS(int\*, int\_ptr\_compare)

TO\_STRINGS(int\*, int\_ptr\_to\_string)

CLONES(int\*, int\_ptr\_Clone)

/\*Mocks Section\*/

MOCK\_FUNCTION(void, hello)

printf("Hello!\n");

MOCK\_FUNCTION\_END(void, )

MOCK\_FUNCTION(void, goodbye)

printf("Goodbye!\n");

MOCK\_FUNCTION\_END(void, )

MOCK\_FUNCTION(void, manyArgs, char\*, name, int, value, float, anotherValue, double,

thirdValue)

printf("Got:\n\tName: %s\n\tValue: %d\n\tVal2: %.02f\n\tVal3: %.02f\n", name, value,

anotherValue, thirdValue);

MOCK\_FUNCTION\_END(void, )

MOCK\_FUNCTION(void, charArg, char, letter)

printf("Got: %c\n", letter);

MOCK\_FUNCTION\_END(void, )

MOCK\_FUNCTION(int, hi, char\*, name, int, num)

for (int i = 0; i < num; i++){

printf("Hello, %s!\n", name);

}

MOCK\_FUNCTION\_END(int, 0)

MOCK\_FUNCTION(void, int\_ptr\_function, int\*, argument)

printf("My int is: %d\n", \*argument);

MOCK\_FUNCTION\_END(void, )

/\*End Mocks Section\*/

END\_MOCKS

void foo(void)

{

manyArgs("Andy", 2, 3.0, 3.14159);

charArg('f');

}

void int\_foo(void)

{

int x = 100;

int\* myIntPointer = &x;

int\_ptr\_function(myIntPointer);

}

CTEST\_BEGIN\_TEST\_SUITE(MicroMock\_C\_Test\_Suite)

CTEST\_FUNCTION(FOO\_TEST\_WITH\_CALLS\_OUT\_OF\_ORDER\_AND\_UNORDERED\_COMPARISON\_PASSES)

{

//Arrange

STRICT\_EXPECTED\_CALL(charArg('f'));

STRICT\_EXPECTED\_CALL(manyArgs("Andy", 2, 3.0, 3.14159));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(UnorderedComparison);

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_CALLS\_OUT\_OF\_ORDER\_AND\_ORDERED\_COMPARISON\_FAILS)

{

//Arrange

STRICT\_EXPECTED\_CALL(charArg('f'));

STRICT\_EXPECTED\_CALL(manyArgs("Andy", 2, 3.0, 3.14159));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_CALLS\_IN\_ORDER\_AND\_UNORDERED\_COMPARISON\_PASSES)

{

//Arrange

STRICT\_EXPECTED\_CALL(manyArgs("Andy", 2, 3.0, 3.14159));

STRICT\_EXPECTED\_CALL(charArg('f'));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(UnorderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_CALLS\_IN\_ORDER\_AND\_ORDERED\_COMPARISON\_PASSES)

{

//Arrange

STRICT\_EXPECTED\_CALL(manyArgs("Andy", 2, 3.0, 3.14159));

STRICT\_EXPECTED\_CALL(charArg('f'));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_IGNORED\_ARGS\_AND\_INVALID\_PARAMETERS\_WITH\_UNORDERED\_COMPARISON\_PASSES)

{

//Arrange

IGNORE\_ARGS(STRICT\_EXPECTED\_CALL(manyArgs("Andrew", 2, 3.0, 3.14159)), 1);

IGNORE\_ALL(STRICT\_EXPECTED\_CALL(charArg('f')));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(UnorderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_IGNORED\_ARGS\_AND\_INVALID\_PARAMETERS\_WITH\_ORDERED\_COMPARISON\_PASSES)

{

//Arrange

IGNORE\_ARGS(STRICT\_EXPECTED\_CALL(manyArgs("Andrew", 2, 3.0, 3.14159)), 1);

IGNORE\_ALL(STRICT\_EXPECTED\_CALL(charArg('f')));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_INVALID\_PARAMETERS\_AND\_UNORDERED\_COMPARISON\_FAILS)

{

//Arrange

STRICT\_EXPECTED\_CALL(manyArgs("Andrew", 2, 3.0, 3.14159));

STRICT\_EXPECTED\_CALL(charArg('f'));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(UnorderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_INVALID\_PARAMETERS\_AND\_ORDERED\_COMPARISON\_FAILS)

{

//Arrange

STRICT\_EXPECTED\_CALL(manyArgs("Andrew", 2, 3.0, 3.14159));

STRICT\_EXPECTED\_CALL(charArg('f'));

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_INVALID\_PARAMETER\_WITH\_USER\_DEFINED\_TYPES\_AND\_UNORDERED\_COMPARISON\_FAILS)

{

//Arrange

int f = 99;

int\* intptr = &f;

STRICT\_EXPECTED\_CALL(int\_ptr\_function(intptr));

//Act

int\_foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(UnorderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_INVALID\_PARAMETER\_WITH\_USER\_DEFINED\_TYPES\_AND\_ORDERED\_COMPARISON\_FAILS)

{

//Arrange

int f = 99;

int\* intptr = &f;

STRICT\_EXPECTED\_CALL(int\_ptr\_function(intptr));

//Act

int\_foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison)

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_ACTUAL\_CALLS\_AND\_NO\_EXPECTED\_CALLS\_AND\_ORDERED\_COMPARISON\_FAILS)

{

//Act

foo();

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison);

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_FUNCTION(FOO\_TEST\_WITH\_EXPECTED\_CALLS\_AND\_NO\_ACTUAL\_CALLS\_AND\_ORDERED\_COMPARISON\_FAILS)

{

//Arrange

STRICT\_EXPECTED\_CALL(manyArgs("Andrew", 2, 3.0, 3.14159));

STRICT\_EXPECTED\_CALL(charArg('f'));

//Assert

COMPARE\_EXPECTED\_AND\_ACTUAL\_CALLS(OrderedComparison);

RESET\_EXPECTED\_AND\_ACTUAL\_CALLS

}

CTEST\_END\_TEST\_SUITE(MicroMock\_C\_Test\_Suite)