# Memory Leak Detection

Mocking is an effective method of finding most memory leak issues within a single module. If done carefully, it can also handle situations where there are modules that create objects and other objects which destroy them… though these are clearly more challenging. By being explicit about when memory is allocated and freed, you can track all the heap memory in your system. This requires significant human focus to test appropriately.

We'll describe a method of unit testing memory here. It promotes modules which completely handle their own memory, because this keeps them unit testable. As soon as this barrier is stretched, however, and the allocation, reallocation, or deallocation of memory is influenced by other modules, then unit testing quickly becomes insufficient for actually tracking the memory of a system.

Unit testing is only a valid stand-in for memory tracking tools in the most straightforward cases.

## Preparing to Mock

Start by creating a header file like "`memory.h`". Inside, we'll put a function prototype for all the memory handling functions we plan to use. For example, we might include something like this:

```
void* calloc(size_t nitems, size_t size);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
void free(void* ptr);
```

If the target compiler's stdlib uses macros or inlines for any of these functions, then it's not going to be sufficient to have the release code pointing at stdlib, and then link to our implementation… there would be two different copies (the mock AND the real version in stdlib). When this happens, you should update the release code to point at `memory.h` instead of stdlib and add a condition compilation like so:

```
#ifdef TEST
void* calloc(size_t nitems, size_t size);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
void free(void* ptr);
#else
#include <stdlib.h>
#endif
```

It may also be necessary to define the type `size_t` in this file.

Using the header above, a release module will pull in stdlib (and therefore the real version of the code). A test will see the prototypes for these functions, and expect us to implement those functions (which we will, using CMock).

## Mocking

The test will then include “`mock_memory.h`”, suggesting that CMock should mock our `memory.h` header and create a mock of those functions. From there, our test is in control. At the most basic level, you can use `Expect` calls to verify that your `malloc` and `free` calls are balanced, but since the purpose of `malloc` and friends is to return memory, this quickly becomes insufficient.

Luckily, when mocking, `ExpectAndReturn` allows us to return a pointer to whatever we want. When we’re dealing with simple tests that need to allocate a known amount of memory, we can get away with return a pointer to static objects. For example, maybe we know that our test is allocating memory for a struct:

```
void test_check_some_memory(void)
{
    struct BIG_FAT_STRUCT_T mem;

    malloc_ExpectAndReturn( sizeof(BIG_FAT_STRUCT), &mem );

    do_stuff();
}
```

In the example above, we happen to know the struct that we’re dealing with, which makes this straightforward. In fact, sometimes we won’t know how much memory we’re dealing with. In those cases, we can often allocate a large buffer to borrow memory for each test. We can count on using it all over again in the next test. Often, when we push things this far, we will need to use stubs to handle the allocation and to verify that the same pointers are freed later.

Keep in mind that mocks can easily test cases where we want to simulate running out of memory (return a NULL!). We have full control when memory interfaces are mocked.

Is mocking always the right answer for unit testing a module that uses malloc and friends?

No.

As we mentioned earlier, there are times where we don't want to unit test memory at all. In those cases, feel free to use the real stdlib functions in your test.

There are also times where *some* of our tests might need to check for different memory handling scenarios and *others* want to focus on the main functionality without worrying about the logistics of internal memory. In those cases, keep in mind that you can have two test files for a single module… one that uses the mock version and one that uses real memory handling functions. This is a powerful method of testing a module at multiple levels.

In any case, memory tracking is all about details.

Happy Testing!