EMBEDDED TESTING

A Book For Those Who Code C And Want Awesome Well-Tested Products WITH UNITY & CMOCK Using Free Tools And For Those Who Enjoy Short Books With Long Titles • by Mark VanderVoord • 2010 •

This Mini-Book attempts to show how the tools CMock and Unity can be used to unit test C projects. These tools were built because Test Driven Development was tedious without some automation and nothing like them existed for embedded development. There is really no reason that these tools are limited only to embedded software, nor any reason that you must do Test Driven Development to use these tools for testing... but we'll think you're extra cool if you do.

It would be seriously Awesome if we could fill 40 pages with cartoons, and you would then be a code-fu master and ready to use all the techniques and tools involved. It would also be cool if we could put all this information on an embedded processor which could be plugged into your brain, giving you instant access to our combined knowledge. Since neither of these are very likely, feel free to post to the forums for either project.

CMock & Unity
Primary Developers

Mike Karlesky
Mark VanderVoord
Greg Williams

*

Original Framework Creator

Bill Bereza

*

Writing, Art, & Design of this Mini-Book

Mark VanderVoord

*

Also Thanks To

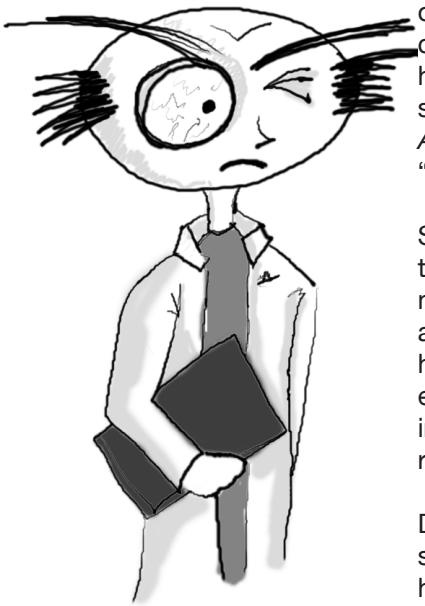
All who have submitted patches, ideas, and questions, especially Martyn Jago

*

Images not created by Mark
(Creative Commons Licensed)

Jason Rogers (x-ray on pg11)

You've started reading this book, which means you like open source tools, are interested in Agile development, write code for embedded software, work for an evil genius, are an evil genius yourself, or possibly all of the above. Well, you're not alone… clearly there are others out there who… at a minimum… work for evil geniuses or *are* evil geniuses.



This is a story about one evil genius, possibly quite like yourself. (otherwise what's the point?)… the infamous Dr. Surly.

Dr. Surly is… well surly… because all the awe-inspiring villain names (Dr. Horrible, Dr. Doom, Dr. Evil) have been trademarked well before he was even born.

He knows it is his destiny to be great… a giant… a pillar of… er, darkness, in the community of evil-doers. But how will he ever accomplish such a feat without an *Awesome* (yes, with a capital "A") villain name?

So his surliness has festered through the years, growing nasty tendrils and tentacles around his brain. His surliness has spread like a tumor, until it exploded as a dastardly plan in his head. He will exact his revenge upon the *entire* world.

Dr. Surly is no idiot. He has seen other villains fail before him. He has witnessed their glorious plans dissipate due to tragic unforeseen flaws. He has watched them fail over and over and over and over again.

But not Dr. Surly. No… Dr. Surly knows what to do. He knows good practices and knows how to make things happen. He will build the ultimate program to take over the world, and it will be tested and *foolproof*! No bug will dare stand in his way… nor will his meager budget. No unexpected behaviors will unfurl tiny banners of defeat… nor will using C keep him from injecting serious doses of Awesome into his work.

Muwahahahahahahah!

A quick search on the internet and he wrings his hands with glee. Yes, he has found it. A simple unit testing framework for C, Unity. It's been used with gcc, IAR, Green Hills, even Visual Studio... This will surely help him.

unity.sourceforge.net

It's site is kinda lame, but otherwise it seems a good tool.

“Ah well... take solace little test framework, I will seek vengeance against those who have wronged you this way... work with me, and you will achieve vengeance!”

Dr. Surly peruses the website, learning about the simple little framework and his options. Does he download the cutting edge release with subversion or just take the latest stable release? Either would work, but he opts for stability. Dr. Surly’s plans are dastardly enough without taking on additional risk.

He downloads the latest release and reads the docs. He is relieved to find that Unity is pure C. He’s a C man... he doesn’t mind letting the included Ruby tools add some *Awesome*... but he’s not interested in learning another language. Let’s see... 1.9.2 is recommended? He goes to *ruby-lang.org* and downloads the one-click installer.

“Let’s get you running, little friend.”

Dr. Surly knows that his project will grow over time... But he also believes in doing *‘The Simplest Thing That Could Possibly Work.’* For now, he won’t worry about setting up dependency tracking for his project or other such frivolities (even though the gruesome complexity of such things make him giddy).

Being the careful sort, Dr. Surly lays out a little map of the directory structure of Unity, to remind himself of what’s what.

Excellent. He opens a command prompt and types:

```
>make
```

An error!? How could that be? He opens the makefile and looks. Ah... a little tweaking is required:

```
C_COMPILER=SurlyC.exe
SYMBOLS=--defineTEST
```

and then

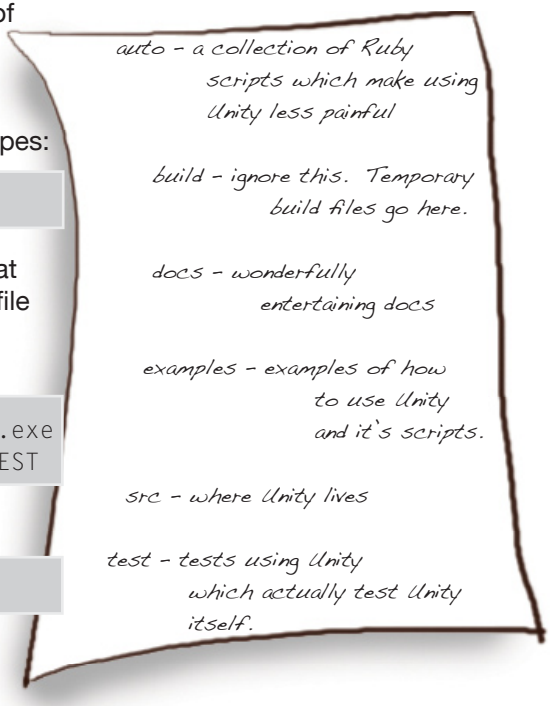
```
>make
```

How exciting! Unity has just tested itself and reported itself to work just fine with his SurlyC compiler.

Wait a minute... the docs say that if he runs the ruby builder, it will also test an example project. He tries it real quick, out of interest. After updating the yaml file, he runs:

```
>rake
```

A quick spattering of tests fill the screen.



Dr. Surly shouts with glee!

His pet sloth, Sourpuss slowly opens his eyes, annoyed that the outburst interrupted a perfectly nice dream... a beautifully recursive dream where he was dreaming about dreaming. Sourpuss casts Surly a withering look, then adjusts his hold on the twig before falling back to sleep.

Dr. Surly ignores the creature, instead thinking of a simple feature... one suitable for familiarizing himself with his new ally. *Eureka!* No Device Of Sufficiently Evil Intent would be complete without a mysterious menacing LED... a *RED LED!*

Great! He starts by using one of the handy scripts to create the module for him. He could create a source file, header file, and test file by hand, but why not just let the script do it?

```
>ruby generate_module.rb MenacingLED

created MenacingLED.c
created MenacingLED.h
created TestMenacingLED.c
```

Glancing at his schematic, he sees that he planned his menacing LED to be on bit 1 of port A. So he'll have to create a function which sets that bit to an output and initializes it to high. He knows it's usually best to test a single module at a time, so he gets to work on that... He'll start by writing one or more tests to describe how that function should work. Updating TestMenacingLED.c, he creates his first test function:

```
#include "unity.h"
#include "MenacingLED.h"
void test_MakeSureLedTurnsOnMenacingly(void)
{
  PORT_A_DIR = 0xFF; //1's are inputs
  PORT_A_OUT = 0x00; //start with all bits 0
  MenacingLed_Init();
  TEST_ASSERT_EQUAL_HEX8(0xFD, PORT_A_DIR);
  TEST_ASSERT_EQUAL_HEX8(0x02, PORT_A_OUT);
}
```

There are also empty functions setUp(void) and tearDown(void) in the generated C file. He leaves those alone for now, though in a more complicated module these may come in handy.

This looks like a pretty good test. He starts by filling memory with data that is clearly not what he wants it to be. He then calls the function to be tested. Finally, he verifies using the TEST_ASSERT statements that everything was configured as it should be. This pattern of setting up, calling the source to be tested, then verifying is a common pattern when writing unit tests.

He needs a few more things before he can run the test. He adds to MenacingLED.h:

```
#include "MicroRegisterDefs.h"
#define MENACING_LED_MASK (2)

void MenacingLed_Init(void);
```

...and a skeleton of the function under test should be enough in MenacingLED.c:

```
#include "MenacingLED.h"

void MenacingLed_Init(void) {  }
```

He goes to the command line and types:

```
>ruby generate_test_runner.rb TestMenacingLED.c
```

He grins a wicked grin as he sees a TestMenacingLEDRunner.c file appear in the output directory. The script scanned his test file for functions of the form *"void test___(void)"* and automatically set up a file to call each of these and collate results.

He then types in the command to have SurlyC compile and link the test, runner, source, and unity files. Then he runs the output executable:

```
>Test

-----------------------
UNITY FAILED SUMMARY
-----------------------
C:\projects\surly\
TestMenacingLED.c:6:
test_MakeSureTheLed
TurnsOnMenacingly:
FAIL: Expected 0xFD was
0xFF.

-----------------------
UNITY TEST SUMMARY
-----------------------
1 Tests 1 Failures 0
Ignores
```

The summary at the bottom of the output shows that a single test was run, as expected. It also shows that this test failed.

Each failure is given more detail in the "Failed Test Summary" earlier in the output, so he reads the item listed there.

It has informed him that line 6 of TestMenacingLED.c failed. Glancing at line 6 of his test, he notices that it contained an assertion to verify that PORT_A_DIR had been set properly. Oh yes! Of course it failed the test! He didn't write the actual source code yet.

Now that he has shown that the test catches a problem with his source code, he writes the minimum amount of code that should make the test pass:

```
#include "MenacingLED.h"

void MenacingLed_Init(void)
{
  PORT_A_DIR &= ~(MENACING_LED_MASK);
  PORT_A_OUT |=  (MENACING_LED_MASK);
}
```

Rebuilding and rerunning the test, he cackles. One step closer to world domination!

```
1 Tests 0 Failures 0 Ignores
```

He's pretty sure that his code is correct, but he remembers that he should verify that none of the other bits are being effected. So, in addition to his existing test, he adds:

```
void test_MakeSureTheLedTurnsOnWithout
          TouchingOtherPins(void)
{
  PORT_A_DIR = 0x00; //opposite other test
  PORT_A_OUT = 0xFF; //opposite other test
  MenacingLed_Init();
  TEST_ASSERT_EQUAL_HEX8(0x00, PORT_A_DIR);
  TEST_ASSERT_EQUAL_HEX8(0xFF, PORT_A_OUT);
}
```

If he was going to have a number of these tests, he could reduce some of the redundancy by creating a helper for the register initialization, the assertions at the end, or both. Since everything is straight C code, he is completely free to create new functions in his test file that can be used to reduce duplication. The only rule is that those functions shouldn't start with the word "test" if you're going to use helper scripts.

```
void
InitializePortA
( uint8 ADir,
  uint8 AOut )
{
  PORT_A_DIR = ADir;
  PORT_A_OUT = AOut;
}
```

Now he can use this function where ever he needs to initialize his ports. This function won't be run by itself as a test because it doesn't start with 'test'.

WAIT! HOW DID HE DO THAT?

This is one of the classic questions about unit testing embedded code: How do you test registers? Ideally, you're not running real hardware here (that's for system tests). You are most likely running either a simulator (getting to use your real compiler) or using a native compiler to compile native test apps (probably faster).

SIMULATORS

This is the easiest situation. Most simulators will let you write to any memory, so you can fill it with something invalid, run your function, then use TEST_ASSERT functions to verify that the contents were updated as you expected. Easy.

NATIVE

Let's say the testing is a native executable on your development machine instead. You can't exactly just write to any memory location you want and assume it's going to be ok (address 0x1234 might be the LED port on your micro, but I bet it's not on your development PC!).

Your goal is to position registers somewhere where it is safe to write and read. You don't want to have to change much of your main code, though, so it's all about how the registers get defined. First, look at the micro's register definition file. Sometimes it's a bunch of structs placed in certain locations... sometimes a bunch of defines. We're going to copy this file and make one that is used only for tests. When you build your release, you'll still use the original, but tests will use this new one. It'll be work, but it's going to make testing so much easier!

For those registers defined in structs, just remove the location specific part of the definition so that the linker will just create it somewhere.

When you're dealing with a pointer de-reference define, replace it with an actual variable... it'll then get mapped somewhere safe by the linker.

```
#define UART3
(*(UART_T*)0x3000)
#define P3 (*(unsigned
short*)0x4000)
```

so our test version looks like:

```
UART_T UART3;
unsigned short P3;
```

NON-STANDARD C

Some compilers have some extra "goodies" where they have support for a specific bit of that port... It's a shorthand and it's not really C.

```
P3 |= 0x01; //normal C
P3_0 = 1; //not normal
```

That's very nice of them... but harder to test with a normal C compiler (like gcc). There are a couple of options:

1. Avoid the bit-sized operators and stick with entire ports.

2. If you really want to use those bit-operators, it's best if you ALWAYS use them, instead of mixing and matching between P3 and P3_0 uses. You're going to create a full byte-size variable for every single one of those bits. Remember, only in your test:

```
unsigned char P3_0;
unsigned char P3_1;
```

Basic Assertions

TEST_ASSERT_TRUE(condition)
TEST_ASSERT(condition)
evaluates code in condition and fails if it evaluates to false
TEST_ASSERT_FALSE(condition)
TEST_ASSERT_UNLESS(condition)
evaluates code in condition and fails if it evaluates to true.

Structs and String Assertions

TEST_ASSERT_EQUAL_STRING(E,A)
TEST_ASSERT_EQUAL_MEMORY
compares two blocks of memory using memcmp. It takes args (E, A, L) where L is the length (you could use sizeof)

Bitwise Assertions

TEST_ASSERT_BITS(M, E, A)
compare two numbers, only bits marked 1 in the mask
TEST_ASSERT_BITS_HIGH(M, A)
evaluates true if all 1 bits in mask are 1 in the actual data
TEST_ASSERT_BITS_LOW(M, A)
evaluates true if all 1 bits in mask are 0 in the actual data
TEST_ASSERT_BIT_HIGH(b, A)
evaluates true if bit is high
TEST_ASSERT_BIT_LOW
evaluates true if bit is low

Automatically Ignore or Fail

TEST_IGNORE( )
TEST_FAIL( )

Integer Assertions

All of the following compare types of integers. They vary in how the data is displayed to the user on failures.

TEST_ASSERT_EQUAL_INT(E, A)
TEST_ASSERT_EQUAL_UINT
TEST_ASSERT_EQUAL_HEX8
TEST_ASSERT_EQUAL_HEX16
TEST_ASSERT_EQUAL_HEX32

These assertions verify that the number actual number is within plus or minus delta of the expected value.

TEST_ASSERT_INT_WITHIN(D, E, A)
TEST_ASSERT_UINT_WITHIN
TEST_ASSERT_HEX8_WITHIN
TEST_ASSERT_HEX16_WITHIN
TEST_ASSERT_HEX32_WITHIN

Float Assertions

(if you have them enabled)

TEST_ASSERT_EQUAL_FLOAT(E,A)
TEST_ASSERT_FLOAT_WITHIN(D,E,A)

Pointer Comparisons

TEST_ASSERT_EQUAL_PTR(E, A)
TEST_ASSERT_NULL(A)
TEST_ASSERT_NOT_NULL(A)

_MESSAGE
add this to any of the assertions to pass in an extra string argument at the end when failures are printed.

_ARRAY
add this to any of the int or float assertions to test an entire array of values. The args are then (E, A, N)

E - expected value
A - actual value
D - delta
M - mask
N - num elements to check

Looking through the list of assert macros, Dr. Surly notices that the floating point asserts are optional. This makes a lot of sense to him... Unity grew up amongst embedded applications, many of whom don't have floating point support. It would be sad if Unity broke just because the compiler didn't know what a 'float' was.

It appears that he can disable floating point support by adding a single define: UNITY_EXCLUDE_FLOAT. Probably the easiest method would be to add the define as one of the command line options, but any method should work.

This makes Dr. Surly curious, though (Evil Geniuses are almost universally curious. They also have compulsive need to monologue at inconvenient times... but that's not particularly relevant at the moment). Anyway, Dr. Surly is curious about Unity's options now. What else can he customize?

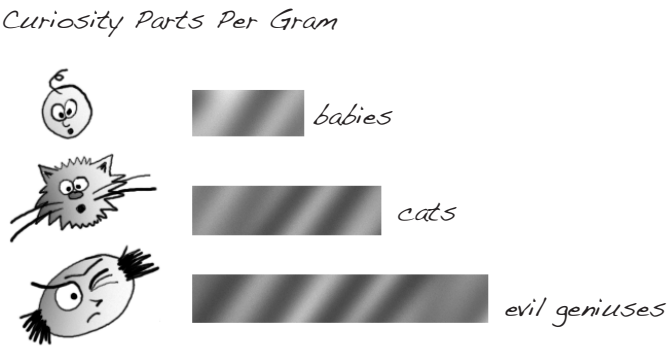
UNITY_INT_WIDTH can be used to specify the number of bits that make up an INT. C is a crazy language, sometimes... Some targets have 16 bit ints, some 32... Unity defaults to 32 bits, but it also supports 16.

UNITY_POINTER_WIDTH can be used for when you want to do pointer comparisons. This is particularly helpful if you have 64 bit pointers instead of the default 32.

UNITY_FLOAT_TYPE is usually defined to be *float*, but you can change it to *double* or whatever wacky floating point types your compiler might support. Why float? Again, Unity grew up with embedded developers, whose systems are single precision.

UNITY_FLOAT_VERBOSE can be defined to make Unity print the expected and actual values for floats, similar to the way integers failures are reported. Ordinarily the float reporting is turned off to avoid bringing in heavy functions like printf. If you've got the resources, though, add this define for useful output.

Everyone knows that you can't compare two floats for equality directly, so TEST_ASSERT_EQUAL_FLOAT doesn't do that. Instead, it checks to make sure that the two values are within a delta of each other. The delta is the expected value times UNITY_FLOAT_PRECISION, which is 0.00001 by default.



UNITY_SUPPORT_64 can be defined to enable 64-bit comparisons. You will also want to define UNITY_LONG_WIDTH to 32 or 64 to give Unity a clue how to get 64-bit support.

You can even change the types of internal variables. Both are unsigned shorts by default... which works for most applications. But if you have an insanely huge file or many tests in a single file (65535 lines or 65535 individual test functions, respectively), you can adjust either UNITY_LINE_TYPE for the number of lines or UNITY_COUNTER_TYPE for the number of tests and failures. If you're a memory miser and can get away with less than 255 of these, you can save a tiny bit of memory by lowering these down to unsigned chars. Setting these values might also help you if you run into memory alignment issues with some targets.

UNITY_OUTPUT_CHAR(a) is a macro that you can use to change how Unity outputs data. Ordinarily, it uses putchar to output to stdout. This works for many applications. A test run as an app can just dump result data to stdout and you can look at the output yourself, pipe it to a file, or whatever you might need to do. Similarly many simulators will forward their stdout to the calling environment. But maybe your simulator doesn't support this or you have to run your code on actual hardware... for those with issues like these, this macro will be your friend.

RESULTS

Since we're on the subject, what is the best way to gather the results from Unity? That depends on two things:

What do you need?

If you're just manually running these tests, dumping all the results to stdout is probably sufficient. Often, automated systems like Continuous Integration servers are flexible enough to handle this too. If you're using Unity's automagically generated test runner, your test exe will return the number of failures as the exit code (0 being that they all pass, and the number saturates at 255 to avoid platform issues). If you need something fancier, maybe that UNITY_OUTPUT_CHAR macro needs some attention.

What can your target handle?

Sometimes your simulator might not allow you to capture its output or other such difficulties. It might be time to consider testing using gcc and saving your real compiler for the release build.

ANATOMY OF A TEST

Tests are a bunch of functions and files that have been splatted together and executed. Here's a quick overview of those parts.

Source File
One of the things built into this mix is the source file that you want to test. Instead of linking this to the rest of your source, though, you're linking it to a test file and some supporting modules, allowing you to just verify that this source file works as you expected. Do that for each source file in your system, and you're Unit Testing.

Test File
The test file contains a bunch of functions in the form of void test_blah(void). Each function that matches this form will be executed as a test. It also contains a function setUp(void) and tearDown(void). The setUp function is run before each test, and the tearDown function is run after each test. You can use these to make sure any shared variables are in a known state, memory is cleared, etc.

Unity
Unity mostly consists of ASSERT macros which you can use to verify that the results of your function calls are as you expect. You could use TEST_ASSERT_TRUE for everything, but there are many more options which will output more helpful information on errors.

Test Runner
If you're using the helpful Ruby scripts, you'll have a Test Runner automatically generated for you. It looks through your test file, finds all the functions that look like tests, and creates a main( ) function for you. This function manages all the calls to tests, setUp, and tearDown. If you're not going to use the scripts, check out the page on Anti-Scriptites.

Executable
Take the source file(s), test file, unity, and test runner and compile each. Link their object files and you've got a test exe for this test file. You'll end up making an exe for each test file you have.

Project Structure
A good way to layout a project is to have all the tests in an adjacent directory... maybe even call that directory 'test' and do something crazy like name the test file for source MyCode.c something like TestMyCode.c.

If you like that sorta thing, you might like some of the other helpful scripts that come with this project. The script to generate a source module will create a C file, header file, and test file from a base name that your provide. There are others that will do similar things but match some design patterns... someone is bound to bring up design patterns in this book somewhere.

Software developers come in all shapes and sizes. Since we're mostly geeks, most of those shapes and sizes aren't particularly attractive, but we make up for that by being Awesome. The only trouble is, the definition of Awesome may vary from person to person. For example, some developers believe Awesome is being in complete control and understanding every detail. Let's call them the Anti-Scriptites. Then there are the Script-O-Rama-Lings, who have the goal of avoiding as much of the tedious work as possible and concentrating on the real features. Here's some thoughts on how Unity can work for either of them.

A N T I - S C R I P T I T E S !

You don't need to use any of the supplied scripts to use Unity. You will, however, need to manually create a runner for your tests, calling each one. Set up a *main* function for each test file to actually call all those crazy test functions. It's easiest if you put main together something like this:

```
void main(void) {
  Unity.TestFile=__FILE__;
  UnityBegin();
  RUN_TEST(test_init);
  RUN_TEST(test_set);
  RUN_TEST(test_three);
  UnityEnd();
}
```

That RUN_TEST thing is a macro defined by Unity. If you're doing a lot of custom handling with your tests (adding code coverage, mocking frameworks, etc) you might need to redefine that. Just make sure it is defined before the include of Unity.h, and it'll use your version instead. You might want to look at the standard implementation first, though... That TEST_PROTECT is crucial, unless you enjoy a good segfault.

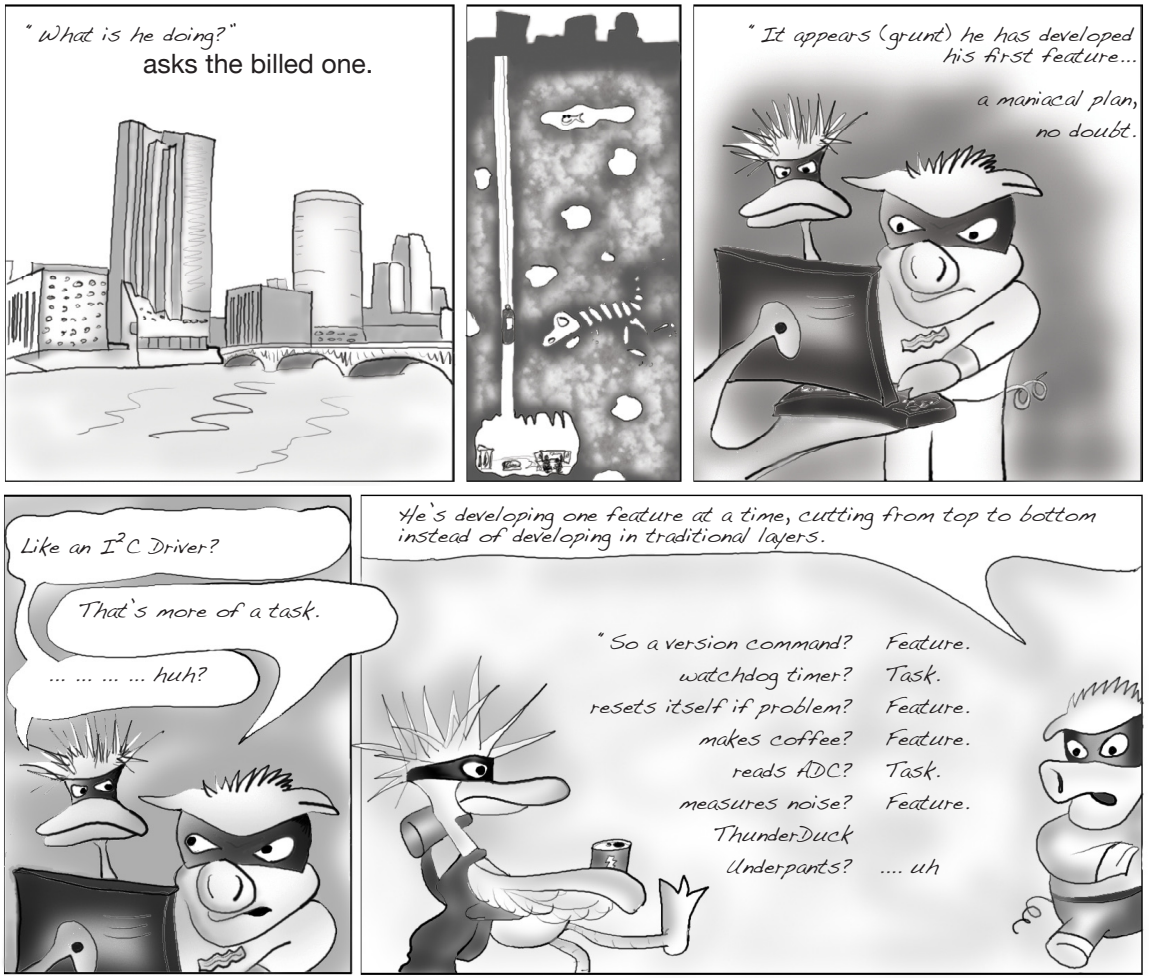
S C R I P T - O - R A M A - L I N G S !

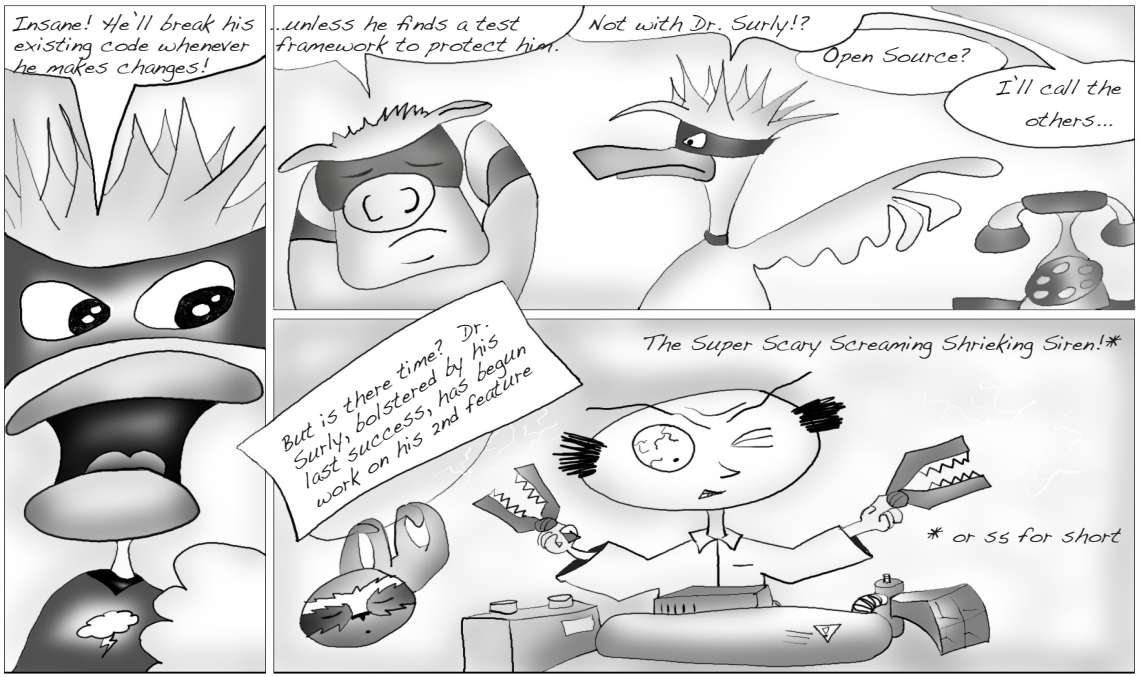
Those scripts that come with Unity aren't bad... but you're still left collecting a test file, runner, and supporting files together, compiling, linking, running, collecting results, then repeating with the next file... whew! Tedious! If only there were a way to just throw files in standard directories and let the magic of dependency tracking just do the rest. You could sit back and tweak yaml:

```
:project:
  :build_root: projects/mine
  :use_exceptions: FALSE
  :use_mocks: TRUE
:paths:
  :test:
    - test
  :source:
    - src
  :include: []
:tools:
  :test_compiler:
    :executable: gcc
```

If this sounds nice, you might consider Ceedling, a build and test framework using Ruby and Rake. Point your web browser to ceedling.sourceforge.net.

Meanwhile, hundreds of miles away, we find a bustling metropolis. Far, far below the city lies a secret bunker. In this secret room, we find two lone figures staring at a computer screen. The short pink one grunts a curious grunt, his chubby hooves poking at the keys.





He already has all the s5 hardware hooked up... he just needs to drive the onboard Timer-Counter peripheral to generate a horribly painful frequency... and maybe scan a digital port to use as an on/off control.

This is a bit more complex than his menacing LED. He would really like to break it into multiple modules... maybe a sub-module for button scanning, one for frequency generation? Something like that.

Testing the Button module or the Frequency module seems simple... it would work like his LED module. But what about his S5 module? It mostly calls those other two... how does he test that?

While he ponders, he hears a little voice, “CMock.”

“Excuse me?” asks Dr. Surly, startled. He looks around the lab, but only finds Sourpuss napping between the beakers.

“I think,” says the voice, “You should check out CMock.”

Dr. Surly stares at the screen, “Unity? Are you talking to me?”

"Yes," responds the framework.

Dr. Surly wonders if perhaps he has been working too long, but jumps onto the internet anyway. A moment later, he has downloaded the latest release of CMock from *cmock.sourceforge.net.*

"Yes," Dr. Surly laughs, "This is exactly it, Sourpuss," (who ignores him), "This tool will automatically create Mocks for me."

When the sloth doesn't respond, Dr. Surly assumes he needs further clarification, "Mocks are fake versions of an entire module. I have a module I want to test, like S5. I have modules that it interacts with, like Frequency and Buttons. Instead of testing them all together, I can test S5 by linking it to Mock versions of the other two. With the Mocks I can verify that functions get called when I expect them to, that they were called with the arguments I desired, and I can even use them to return whatever results I want to test!"

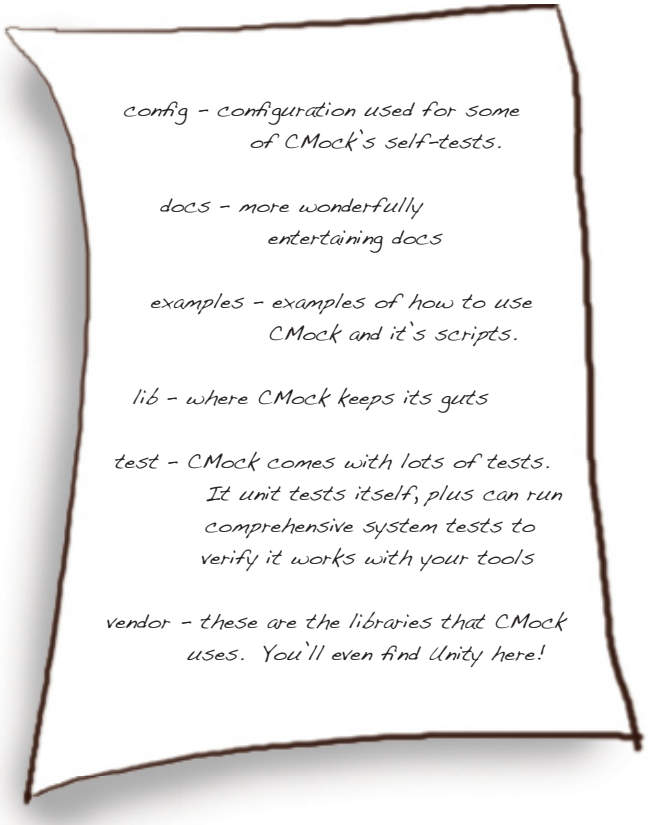
Sourpuss continues to stand firm with apathy.

"Don't you see? This tool is going to allow me to break my code down naturally without having to get it all working at once! Plus, it'll be fully tested!"

The sloth opens one eye, briefly, then yawns and returns to sleep.

"On my own, I see," Dr. Surly grumbles.

"Hardly," says a pair of voices from the computer.



Dr. Surly starts by creating an S5 module with a script call:

```
>ruby generate_module.rb S5
```

He cracks open the test file and begins to insert tests. The CMock manual said to set up all expectations before calling the function being tested. He wants to make sure that the s5 is silent when the switch is off (no sense deafening himself). So, he creates a test:

```
#include "MockButtons.h"
#include "MockS5Ctrl.h"

void test_S5_Exec_ShouldBeSilentWhen0(void)
{
  Buttons_CheckS5Switch_ExpectAndReturn(0);
  S5Ctrl_Silence_Expect();

  S5_Exec();
}
```

Yes, something like that. He expects that the s5 switch will be checked, and it will return 0 for the purpose of this test. Seeing the 0, the exec function should then call for silence. Then the test runs the actual function to verify that all the expectations have occurred.

He doesn't need to make the Mock header files himself, CMock will handle that for him. But he does need to make sure that they are included in the test (which he has done) and he needs to ensure that the real headers exist and have proper prototypes.

He uses the scripts to create a couple more modules (Buttons.c and S5Ctrl.c). He puts a prototype for Buttons_CheckS5Switch and S5Ctrl_Silence in the correct headers. He also puts a prototype for S5_Exec in S5.h, and an empty implementation in S5.c. That should be enough to run a test. He lets it compile, link, and run. The test returns a single failure, described as "Function 'Buttons_CheckS5Switch' called less times than expected."

Well that makes sense. Since S5_Exec() is just an empty function so far, the test has correctly informed him that he had expected Buttons_CheckS5Switch to be called, but that it was never was. It quits each individual test at the first sign of trouble, so it hasn't yet noticed that S5Ctrl_Silence also hasn't been called.

Well, now that he has a failing test, it must be time to write enough code to make it pass. He turns his attention to his S5 module. In particular, Dr Surly writes the S5_Exec function, ignoring the fact that Buttons_CheckS5Switch and S5Ctrl_Silence still do not actually exist.

The exec function should be simple enough. It needs to check the switch to see if the level is low (zero). If so, it should configure the system for silence. Easy enough. He types:

```
void S5_Exec(void)
{
  if (Buttons_CheckS5Switch() == 0)
  {
    S5Ctrl_Silence();
  }
}
```

He reruns the test, this time seeing that everything passes. OK, so next he would like an extremely loud screech at 20KHz when the button is pressed. So, he adds a test for that:

```
void test_S5_Exec_ShouldBePainfullyLoudWhenSwHigh(void)
{
  Buttons_CheckS5Switch_ExpectAndReturn(1);
  S5Ctrl_SetFrequency_Expect(20000);
  S5Ctrl_Loud_Expect();

  S5_Exec();
}
```

He grumbles with annoyance when he hits compile and is reminded that he needs to add a prototype of S5Ctrl_SetFrequency(Frequency) and S5Ctrl_Load() to S5Ctrl.h.

Dr. Surly is persistent, though, and knows it will become second nature. He adds the prototypes and nothing else. Running the tests, he sees the expected failure: “Function ‘S5Ctrl_SetFrequency’ called less times than expected.” The exec function hasn’t been designed to actually call the new functions yet.

He updates his function to support the new feature:

```
void S5_Exec(void)
{
if (Buttons_CheckS5Switch())
  {
    S5Ctrl_SetFrequency(20);
    S5Ctrl_Loud();
  }
  else
  {
    S5Ctrl_Silence();
  }
}
```

He runs the test again, expecting everything to be fine... but... what? A failure?

```
Expected 20000 was
20. Function ‘S5Ctrl_
SetFrequency’
called with unexpected
value for argument
‘Frequency’
```

Ah! He goofed up the units. He wanted 20KHz (or 20000 Hz), not 20Hz!

He corrects the function to pass 20000, and updates the name of the argument to ‘Hz’ instead of ‘Frequency’ to clear things up in the future.

He also realizes that he really only wants the “loud” setting engaged when the frequency was properly set. He updates the S5Ctrl_SetFrequency function to return the actual frequency. This requires him to change the Expect in his last test:

```
S5Ctrl_SetFrequency_ExpectAndReturn(20000, 20000);
```

He also adds a new test to capture the new behavior. In this test, he expects that S5_Exec will request 20KHz, but his mock of S5Ctrl_SetFrequency will return only 14KHz, enabling him to test what would happen:

```
void test_S5_Exec_ShouldBeSilentIfNotLoudEnough(void)
{
  Buttons_CheckS5Switch_ExpectAndReturn(1);
  S5Ctrl_SetFrequency_ExpectAndReturn(20000, 14000);
  S5Ctrl_Silence_Expect();

  S5_Exec();
}
```

Of course, it fails when he runs the test, because S5_Exec is still ignoring the return value and is calling S5Ctrl_Loud. He updates the function under test:

```
void S5_Exec(void)
{
  if ( (Buttons_CheckS5Switch()) && (S5Ctrl_SetFrequency(20000)) )
      S5Ctrl_Loud();
  else
      S5Ctrl_Silence();
}
```

There. That’s better. Honestly, Dr. Surly is feeling a bit better knowing that he has tested all these conditions in a robust manner.

AN ASIDE ABOUT ZOMBIE HORDES

At one time, zombies were a real problem. They'd stumble into your office and suddenly half your coworkers were brainless (assuming they weren't already)! The only solution was a stick of dynamite or a shotgun, not exactly common in an office.

Thanks to modern technology, those days are past. Each instance of CMock comes with its own tiny zombie horde, carefully trained for your convenience. When you think about it, it's a perfect match. You get the tireless work ethic of the undead while they get to drool over large engineer brains. It's beautiful, really.

Exactly what can you do with a zombie horde? Try this to find out: call CMock from the command line, optionally passing the path to a yaml file containing options:

```
> CMock (-oYamlFile) Files
```

Or choose one of the following forms from Ruby or Rake, then use it:

```
    #uses defaults
m = CMock.new
    #specifies YAML file
m = CMock.new(YamlFile)
    #specifies directly
m = CMock.new(OptionsHash)
    #use it!
m.setup_mocks(FilesToMock)
```

Either way, the zombies start gnawing on the header files, picking out all the function declarations that can be found. But don't worry... they're working for you.

Some zombies are really old (*the ones with more missing body parts*). Old zombies have seen a lot of ugly legacy code and poor libraries. They pride themselves on being able to parse your headers, no matter how ugly the code is. But, if you fear they may need help (*zombies aren't exactly known for their intelligence*), CMock has a lot of configuration options.

They then piece together a Mock (fake) version of each function. Just as zombies are mutated shells of original people, so Mocks are zombified shells of the original functions. On the surface, it looks mostly like the original. Maybe its pallor is more gray... its gait a little crooked. But inside, things are completely different. The insides don't

care about real-life things... instead they care only about simple things. They count the number of times you expect functions to get called, and groan when that doesn't happen. They make sure that the arguments expected are the arguments used, in the correct order. They even queue up return values that can be returned from calls to the Mock, so that the test writer can inject any values desired.

They do all this through _Expect functions. One of these Expect functions is created for each Mock. An ExpectAndReturn is used if the original function returned something, otherwise Expect is used. Later we'll see how plugins will create additional functions.

To deal with all sorts of code, from standard to proprietary, from cutting-edge to legacy, the CMock zombies make some assumptions. Let's look at what these are. You can override many if you desire a different behavior (*zombies are weak-willed at best*).

const - Attributes like 'const' will appear in your Mock just like the original function, but might be dropped from Expects to make testing easier.

extern - Any function declaration which starts with extern is ignored. This is a convention that the zombies have adopted after watching the way large legacy systems tend to throw extra extern'd functions into their header files. If you don't agree, just change :treat_externs from :exclude to :include.

anonymous arguments - C lets you specify a function prototype without the variable names, leaving only types. CMock does its best to recognize these cases, but can't read minds. If you have something like "unsigned doohicky" it guesses that doohicky is the name of something of type "unsigned". It's possible that's not correct. To help a little bit, the config arrays :attributes, :treat_as, and the :unity_helper can give CMock more information about your types (*see Captain Bacon's Savory Sidebar*).

attributes - Some compilers support non-standard attributes like '__irq' or 'input' in functions or arguments. If you add these to the :attributes list, you can give those zombies an edge on parsing.

function pointers - Function pointers (anonymous or otherwise) are ugly. A typedef'd function pointer is as easily dealt with as any other type... Function pointers which appear inline in the function declaration are typedef'd by CMock for easier usage.

void - Zombies treat 'void' as something special, and why shouldn't they? A well-placed void is the difference between having arguments or not. A void distinguishes between functions which get Expect and those that get ExpectAndReturn. Some odd systems out there actually typedef void, like ANTI_ZOMBIE_VOID. Tell the zombies about these using :treat_as_void, they'll work through this subtle subterfuge and get back to work.

debugging tests - Sometimes it's useful to make CMock complain more... The option :memcmp_if_unknown can be set to false to make CMock complain whenever it comes across an unknown type (so you can add it to :treat_as or :unity_helper). The option :when_no_prototypes can be changed from the default :warn to :error to make CMock fail whenever you ask it to make a Mock of a header where no functions are detected (it can also be set to :ignore for silence).

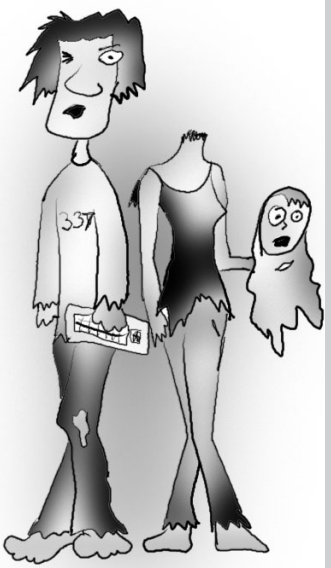
preprocessing - CMock knows a little about C, but it's not a compiler. If your headers contain lots of preprocessor macros and #ifdefs, you might want to consider preprocessing your headers before giving them to the zombies.

NOTE: It should be noted that :plugins and :enforce_strict_ordering can (and should) be passed to generate_test_runner too. A mismatched set of options may lead to compiler errors or (worse) more subtle problems. To make this more convenient, the generate_test_runner script accepts a yaml section of :cmock: if it can't find the standard :unity: section, so you can just give it the same yaml file you're using for cmock.

build customization - You can control where your mocks get placed by setting :mock_path or even what they are named by setting :mock_prefix. These will default to mocks\ and Mock respectively.

Your very own sample yaml config:

```
:mock_path: 'where\mocks\'
:mock_prefix: mocked
:plugins:
  - :cexception
  - :ignore
  - :callback
:includes:
  - mytypes.h
  - mydefs.h
:attributes:
  - __irq
  - __fiq
:enforce_strict_ordering: true
:cexception_include: 'path\
Exception.h'
:treat_as:
  :MY_INT: INT
  :MY_CHAR_STR: STRING
:treat_as_void:
  - CUSTOM_VOID
:memcmp_if_unknown: false
:when_no_prototypes: :error
  #opts :ignore, :warn, :error
:when_ptr: :compare_data
  #opts for last one
  #  :compare_ptr
  #  :compare_data
  #  :compare_array
```



compile-time options

Zombies survive with a lot less brain cells than most of us... kinda like embedded processors. Because of this, you sometimes may want to control how they deal with that limited memory. There are some #defines you can make (as command line options to your compiler, most often) to help:

You can specify if CMock has a fixed amount of memory to work with or if it should pull memory from the heap by defining CMOCK_MEM_STATIC or CMOCK_MEM_DYNAMIC. By default, you get a static block of 32kb.

Then, specify CMOCK_MEM_SIZE. If you're working with dynamic memory, this is the size of a chunk you allocate each time you need to "dip in for more." A larger number makes for higher performance. This number is the total amount of memory CMock will use if you're using static memory.

Don't worry, you'll get a friendly failure with a clear message if the tests attempt to pass this limit.

Many targets get picky about where you access data. You should probably tell CMock about that. You can do that by setting CMOCK_MEM_ALIGN. Check out this handy chart:

0 - Don't align (or align to byte)
1 - Align to 16-bit word
2 - Align to 32-bit word
3 - Align to 64-bit word

CMOCK_MEM_INDEX_TYPE - Internally, CMock has some accounting to do. This is the type it should use an an index. Usually it's set to be an unsigned int... but you can make it something smaller if you're working with little memory.

CMOCK_MEM_PTR_AS_INT - Finally, CMock wants to treat your pointers as numerical types sometimes... and it needs to make sure it's got a big enough int to handle that. On many machines, that's an unsigned int... but sometimes (we're looking at you 64-bit machines), it might be something else.

Thunder Duck touches down lightly on the top of the building, his bill grim. He presses his code-breaking box to the stairwell door and waits a moment. It beeps and the door clicks open. Inside he finds rows and rows of servers. He crimps into one of the lines, then plugs it into his wrist computer... a moment later, it begins to spit out data.

"Captain Bacon... he's definitely Mocking... But there's no way he can stick with standard types forever! When he needs to pass structs or arrays as arguments, his evil plans will be thwarted!"

"No sir... CMock can handle any type you want. If it doesn't know the type, it'll perform a simple memory comparison."

"Ha!" says Thunder Duck, unphased, "He'll be wasting all his time trying to figure out which element was actually wrong."

"True... unless..."

"Unless, what? Spit it out, Bacon... I need to know what I'm up against!"

"Sir, he can add his own assertions to unity helper files. As long as he follows the naming convention, CMock can find them in the specified helper file and it will automatically use any types that it recognizes."

"Blast! Surely that must be incredibly difficult!"

"No sir, if he had a type PIG_T, he would just need to have a macro defined for UNITY_TEST_ASSERT_EQUAL_PIG_T, taking four arguments... the expected value, the actual value, the line number (for printing) and an error message."

"And CMock will scan that, and automatically match the types?"

"Yes sir."

Thunder Duck sighs, "So what about arrays, Bacon? How does it know what he wants? What if it's a null pointer... or just a pointer to a single structure?"

"Pointer treatment is configured by setting :when_ptr. If set to :compare_ptr then only the pointers address is compared."

"But the default configuration is :compare_data. With this setting, if the pointer that you Expect is a NULL, it'll verify that the actual pointer is also NULL. If it's actually pointing to something, then a single element comparison is performed by default."

"Excellent! Just the first element isn't enough to cover all cases!"

"Unless he enables the :arrays plugin. If he does that, every mocked function gets its usual Expect function... plus a new ExpectWithArray function. For each pointer argument passed to that function, there are now two arguments. The first is a pointer to the expected data, and the second is the number of elements to check."

Bacon sends an example to Thunder Duck's high-tech wristwatch, who scans the source code, noticing the BACON_T and FOOD_T types.

"And those," Thunder Duck asks slowly, almost afraid to finish his question, "Those are more custom types that are automatically discovered in the Unity Helper?"

"Yes sir," Captain Bacon says, grimly, "Yes."

Thunder Duck swallows. He needs to take action soon! Dr. Surly can't be allowed to continue with these plans.

Bonus! *Try setting to :smart to have it act like :compare_ptr when you specify 0 elements, or :compare_data otherwise!*

```
void AddBacon(BACON_T* SomeBacon);
void AddBaconToAll(BACON_T* SomeBacon, FOOD_T* Foods);
```

will be Mocked with the following ExpectWithArray functions:

```
void AddBacon_ExpectWithArray(BACON_T* SomeBacon,
             unsigned int SomeBaconDepth);
void AddBaconToAll_ExpectWithArray(BACON_T* SomeBacon,
             unsigned int SomeBaconDepth,
             FOOD_T* Foods, unsigned int FoodsDepth);
```

CAPTAIN BACON's SAVORY SIDEBAR

Just like Captain Bacon, you can add your own custom types. When faced with non-standard types, you have options. Try all three!

0. Do Nothing

CMock will perform a memory compare and report mismatches. The negative: No details.

1. Pseudo Standard Type

```
typedef int FUNKY_NUM;
typedef char* SILLY_STR;
typedef unsigned char* Meh;
```

Just specify how you want them treated in your YAML file. Choose INT, HEX8, HEX16, HEX32, or STRING:

```
:treat_as:
  FUNKY_NUM: INT
  SILLY_STR: STRING
  Meh: HEX8*
```

2. Your Own Swank Types

We could define a macro in a helper function like MyHelper.c for the new type like UNITY_TEST_ASSERT_EQUAL_WACKY_T, then add the path in our YAML file:

```
:unity_helper: tst\
MyHelper.c
```

We find Dr. Surly still working in his lab... a look of elation on his face. A look of elation brought on by simultaneous discovery and security. He has just discovered the option to enforce strict ordering... an option which makes him feel much safer. Instead of just enforcing the order of arguments to a specific function call, it enforces the order of all function calls made.

```
:enforce_strict_ordering: true
```

When Dr. Surly wants to use his Energy Ray, he knows he needs to charge the system three times, each time passing the number of charges remaining. Only then can he fire the Ray. He would obviously set up his test like so:

```
void test_EnergyRay_Fire_ShouldChargeThreeTimesThenFire(void)
{
  EnergyRayHardware_LoadCharge_Expect(2);
  EnergyRayHardware_LoadCharge_Expect(1);
  EnergyRayHardware_LoadCharge_Expect(0);
  EnergyRayHardware_Fire_Expect();

  EnergyRay_Fire();
}
```

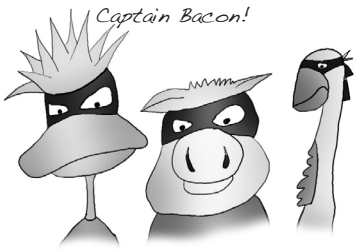
CMock always catches problems like the one below. It always remembers the order of parameters passed to a single expect (notice we're counting in the wrong direction):

```
void EnergyRay_Fire(void)
{
  int i;
  for (i = 0; i < 3; i++)
    EnergyRayHardware_LoadCharge(i);
  EnergyRayHardware_Fire();
}
```

But if this option were disabled, there is a potential error he could make that would go uncaught. He feels safer knowing that he has protected himself from this possibility. The key is that Dr. Surly knows that he needs to charge *before* he has enough energy to fire.

```
void EnergyRay_Fire(void)
{
  int i;
  EnergyRayHardware_Fire();
  for (i = 2; i >= 0; i--)
    EnergyRayHardware_LoadCharge(i);
}
```

He understands that some people want to turn off the global-order-enforce option to make test-writing simpler, but he's not interested in simplicity. He's interested in correctness. That's why he's testing in the first place! As soon as he has enabled :enforce_strict_ordering, problems like this are obvious.



"I've been using this CMock thing while planning our raid on Dr. Surly's lab. I've developed a series of smoke machines which are going to fill the area with smoke so that we can arrive under cover."

"A genius plan," says Captain Bacon.

"Yes, genius," agrees The Gobbler.

"Now, I'm working on an App for our hero-phones to make them vibrate in the correct order to specify when each of us should move... so we all arrive in Surly's lab at the same time."

"Fabulous," says Bacon.

"Fabulous," agrees Gobbler.

"But there's a problem."

"Sir?"

"The function which coordinates phone vibration timing also reloads each smoke machine.... can I just specify vibrate calls without all the reload calls?" the Duck asks, showing them his code.

```
void test_Phones_Should_
VibrateInSequence(void)
{
 Smoke_Reload_Expect();
 Smoke_Reload_Expect();
 Vibrate_Expect(Duck);
 Smoke_Reload_Expect();
 Vibrate_Expect(Bacon);
 Smoke_Reload_Expect();
 Vibrate_Expect(MrGuy);
 Smoke_Reload_Expect();
 Smoke_Reload_Expect();
 Vibrate_Expect(DimBulb);

 CoordinatePhones();
}
```

"Maybe that's an indicator that you should refactor that into two functions."

“Let’s say that’s not possible,” he growls.

“No problem, sir,” the pig replies, typing at the terminal, “Just enable the :ignore plugin, then tweak your test to look like this.”

```
void test_Phones_Should_VibrateCorrectSequence(void)
{
  Smoke_Reload_Ignore();
  Vibrate_Expect(ThunderDuck);
  Vibrate_Expect(CaptainBacon);
  Vibrate_Expect(MrToughGuy);
  Vibrate_Expect(TheDimBulb);

  CoordinatePhones();
}
```

“Nice... so what if Smoke_Reload has a return value?” asks the duck.

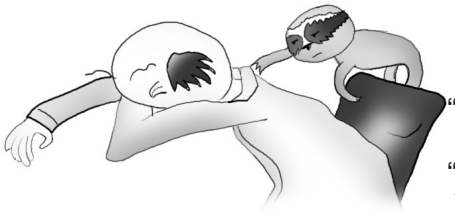
“No problem sir, you can use IgnoreAndReturn and specify just the return value, like this.”

```
void test_Phones_Should_VibrateCorrectSequence(void)
{
  Smoke_Reload_IgnoreAndReturn(100);
  Smoke_Reload_IgnoreAndReturn(80);
  Vibrate_Expect(ThunderDuck);
  Vibrate_Expect(CaptainBacon);
  Vibrate_Expect(MrToughGuy);
  Vibrate_Expect(TheDimBulb);

  CoordinatePhones();
}
```

“Here,” explains Captain Bacon, “We’re telling our test to return 100 the first time Smoke_Reload is called, then 80 every other time. We could have just specified a single value to always be returned, or could specify all the returns values expected... while still ignoring the arguments passed to the function.”

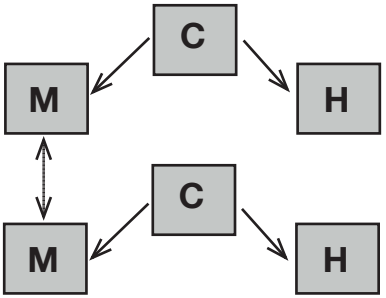
As Dr. Surly begins to think about his next feature, the Clashing Tentacles with Chaotic Tendencies, he realizes that this module will be significantly larger than those he has developed previously. Thinking about testing such a big module seems daunting.

"It's too big! Too complex!" he yells, his frustration seething in his voice. He throws his head on the table.



A calm hand wrests on his shoulder... or... a paw. Looking back, he finds himself staring at the serene face of Sourpuss, whose rarely-open eyes exhibit a calming effect.

Sourpuss steps around him slowly and gingerly takes the pencil. He draws in slow precise strokes in the air. Surly watches over his shoulder.



"Oh! Design Patterns! I've seen some of these before."

"OK, First MCH. Three .c files? Well, yes, I suppose a module isn't necessarily one file... but it's just so... er... uh."

"Why three files?"

"Model, Conductor, Hardware. Yes, I see Sourpuss... All the hardware interface happens in the Hardware file. The Model is the module's only link to the outside world, and contains all private state and data. The Conductor just coordinates them. Ideally the Coordinator reads like simple instructions, like:

```
if (Model_IsReady( ))
  Hardware_SendPacket
if (Hardware_Triggered( ))
  Model_TellSomeone
```

"This explains why the arrows point out from Conductor. Only the Conductor knows about the other parts of the module."

Dr. Surly grabs the paper from Sourpuss and stares at it, "I see, I see... so the Conductors might all get called from the main loop... they conduct the interaction between the Models and the Hardware... and Modules talk through their Models only, right? Very eloquent, Sourpuss... I must say."

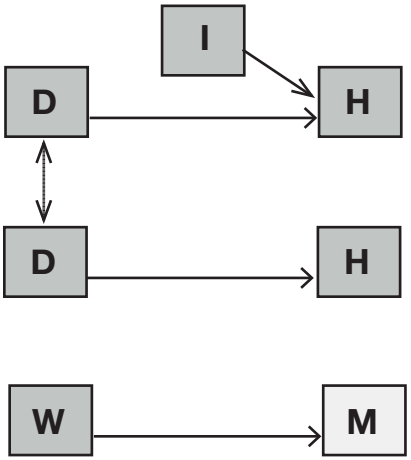
"What happens if I were to use an OS? Ah! Of course... Each Conductor becomes a task. Very nice, Sourpuss.... when do we use this?"

MCH	periodic background tasks and places where you don't need to wait for results
IDH	event driven features and places where we expect a response in a timely manner.
W	when we just want to specialize an existing interface

"Yes, I see that's a nice chart," says Dr. Surly, "but what are these other patterns that you have listed?

"I see, D(I)H is Driver Interrupt Hardware, and the Interrupt is an optional aspect. The main interface and state are stored in the Driver, much like the Model of MCH. Unlike a Model, though, this directly calls Hardware. If there is an interrupt, the hardware portion of the handler is in Hardware as expected, then any data or event flags that need to be stored are performed in a callback function in the Driver. The Driver sets up the callback during initialization."

"You don't need to explain this last one, Sourpuss," says Dr. Surly, "I can guess that a Wrapper is just a thin abstraction around an existing module... like if we want to coordinate a couple of GPIO pins to control motor speed, the MotorSpeed module is likely to just be a wrapper around GPIODriver."



SOURPUSS SLOTH'S SANGUINE SIDEBAR

There are five thousand seven hundred and twenty one design patterns that can be applied to any good embedded application. I know... I painstakingly counted them last Thursday. You could (and most embedded developers do) ignore the world of design patterns completely and design from your own experience and instincts. There might be patterns in there... but you don't care what they are. So why are we spending two precious pages on this? Two reasons, really.

1 - These patterns have proven to be useful for keeping complexity down and for making our software easily testable.

2 - They're good examples of how you shouldn't be afraid to break things into smaller pieces if it means you can test it easier.

What if you wanted to use these patterns, but you have modules that sometimes are waited on and sometimes are polled. Our experience has shown a general rule can help here too. Occam's razor says the simplest answer is usually correct. When you are using an RTOS, the simplest answer is to use MCH (Conductors make such nice tasks). When you're doing the classic 'super-loop' application, DH is simpler because you would otherwise need to kick that Conductor periodically to keep it cranking away. That would be yucky. If you're using coroutines... well... you're on your own. We haven't tried a coroutine application since we learned the rest of this crazy testing stuff.

His last feature, the Clashing Tentacles with Chaotic Tendencies, will flail around wildly, smacking and bludgeoning everything they touch. When one of them manages to grab something, it will pick it up and throw it as far as possible. Anxious to get going, Dr. Surly generates the source module as an MCH triad and digs in.

```
> generate_source_module -pMCH Tentacle
```

If Dr. Surly understood Sourpuss's description of the MCH pattern correctly, he believes that starting with the Conductor makes the most sense. He opens TestTentacleConductor.c and enters his first test:

```
#include "MockTentacleModel.h"
#include "MockTentacleHardware.h"
#include "TentacleConductor.h"

void test_TentacleConductor_Exec_ShouldFlailWildlyWhenFree(void)
{
  TentacleModel_HasSomething_ExpectAndReturn(FALSE);
  TentacleHardware_FlailWildly_Expect();

  TentacleConductor_Exec( );
}
```

That looks pretty good. It asks the Model if anything has been captured. This test injects a FALSE as the answer to this query, so the Hardware should then be instructed to flail wildly. If that doesn't happen, it'll fail the Expect calls.

He adds declarations for TentacleModel_HasSomething and TentacleHardare_FlailWildly to TentacleModel.h and TentacleHardware.h, respectively. Then, he adds a declaration for TentacleConductor_Exec as well as an empty implementation. Running the test, it fails as expected. He then updates the Conductor.

```
void TentacleConductor_Exec(void)
{
  if (!TentacleModel_HasSomething())
    TentacleHardware_FlailWidly();
}
```

The test passes. He now has a choice. Will he continue to implement TentacleConductor, or start filling out details of the Model or Hardware? Whichever he chooses, how does he keep track of what he needs to do? As the panic begins to arise, he is reminded of a unity feature: TEST_IGNORE. He quickly adds a test to the Model and Hardware containing a line like this:

```
TEST_IGNORE_MESSAGE("Don't forget TentacleModel_HasSomething");
```

These ignore messages will show up in the test report but won't count as failures. That should remind him what needs to be done... kinda like leaving a trail of breadcrumbs. So, breadcrumbs in place, he adds another test to the Conductor.

```
void test_TentacleConductor_Exec_ShouldThrowStuffItCaught(void)
{
  TentacleModel_HasSomething_ExpectAndReturn(TRUE);
  TentacleHardware_ThrowIt_Expect();

  TentacleConductor_Exec( );
}
```

This, of course, fails. He updates the source to make it pass:

```
void TentacleConductor_Exec(void)
{
    if (TentacleModel_HasSomething())
      TentacleHardware_ThrowIt();
    else
      TentacleHardware_FlailWidly();
}
```

Happy with the Conductor for now, Surly moves on to the Model. He figures that he'll have a new module called Grip, which will detect if something is in the tentacle, so the TentacleModel can call that. The Grip check is pretty fast and can be performed on demand, so he decides it'll be a Driver-Hardware module. First he creates the new modules:

```
> generate_source_module -pDH Grip
```

Then he adds the first test to TestTentacleModel.c

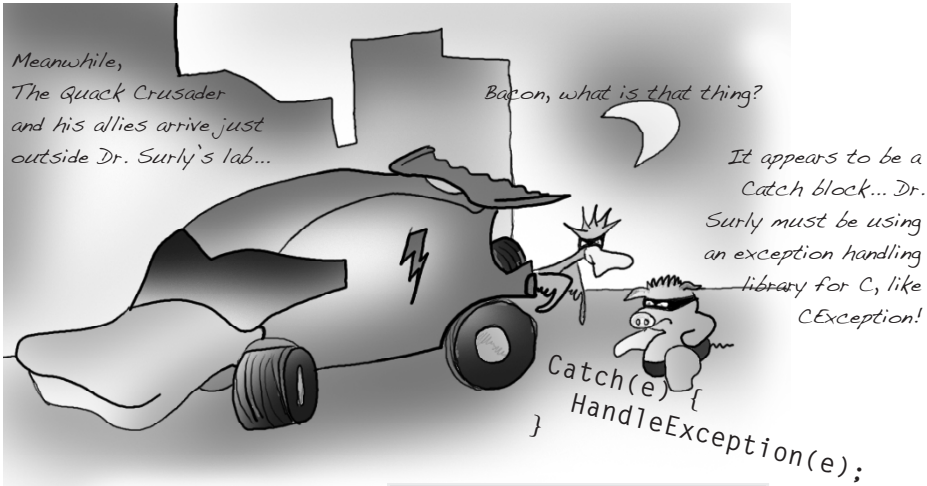
```
#include “MockGripDriver.h”
#include “TentacleModel.h”
void test_TentacleModel_HasSomething_ShouldCheckGrip(void)
{
  GripDriver_HasSomething_ExpectAndReturn(TRUE);
  TEST_ASSERT_TRUE(TentacleModel_HasSomething());

  GripDriver_HasSomething_ExpectAndReturn(FALSE);
  TEST_ASSERT_FALSE(TentacleModel_HasSomething());

  GripDriver_HasSomething_ExpectAndReturn(TRUE);
  TEST_ASSERT_TRUE(TentacleModel_HasSomething());
}
```

He adds a prototype for GripDriver_HasSomething, adds a TEST_IGNORE for it, and adds the empty TentacleModel_HasSomething (is this getting automatic yet?). Run and fail. Implement and refactor (if needed). It’s this pattern over and over again.

Dr. Surly is so engrossed in his work, that he almost doesn’t notice when his proximity alarm begins to blink. When it finally catches his attention, he yells to Sourpuss, “We’ll have to release with the last version! No time to finish the tentacles... let’s crank this thing onto the roof!”

Sourpuss leans over and presses “Deploy” on their continuous integration server. One of the advantages of working this way is that you should have a fully tested system almost every time you commit... so doing a quick release (especially when being invaded by superheroes) shouldn’t be too much of a problem.

As the server hums, Dr. Surly opens another window and attaches to his security cameras. He finds Thunder Duck and his meddling sidekicks near the back door. He should have known it would be them! No matter... his dastardly plans have evolved far enough. It is time.



"What does it do?" asks the duck.

"It's for error handling. Instead of returning error codes from every function or other messy ways of handling errors, CException allows control to be transferred directly from a Throw call to the most recent Catch, where the error can be cleaned up as desired."

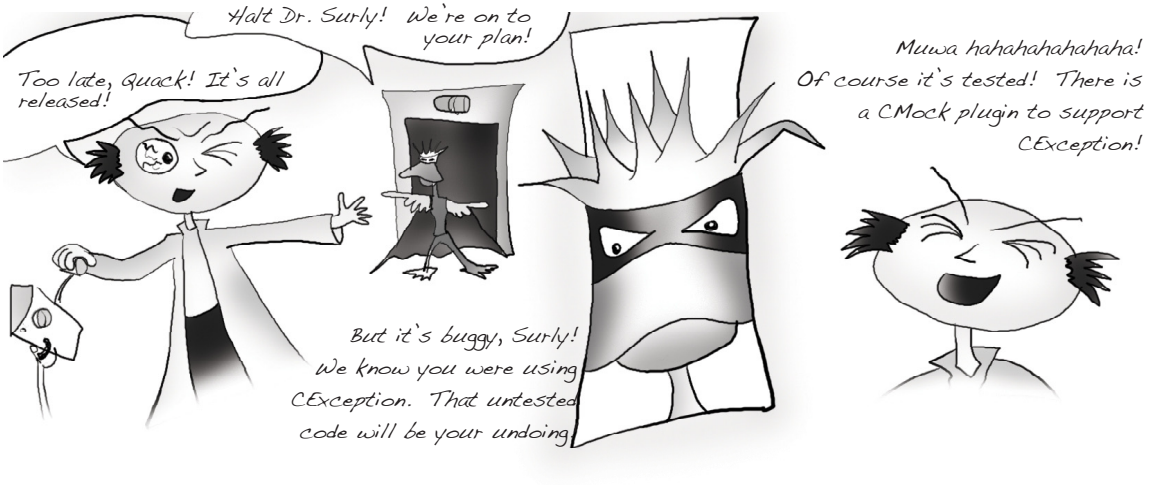
```
void ThisFuncHasError()
{ Throw(0x53); }

void SomeFunc(void)
{
  Try {
    //Any problems here
    //get caught below
    AFunction();
    ThisFuncHasError();
    ThisFuncNotCalled();
  }
  Catch(e) {
    //RespondToErrorHere
    //e has thrown id
  }
}
```

"How does that work?" he asks.

"It uses the standard library setjmp and longjmp functions," says Captain Bacon.

"This is great!" exclaims Thunder Duck.

"This is stack manipulation kinda stuff... there is no way he could be testing that properly. Let's go in there and get him!"

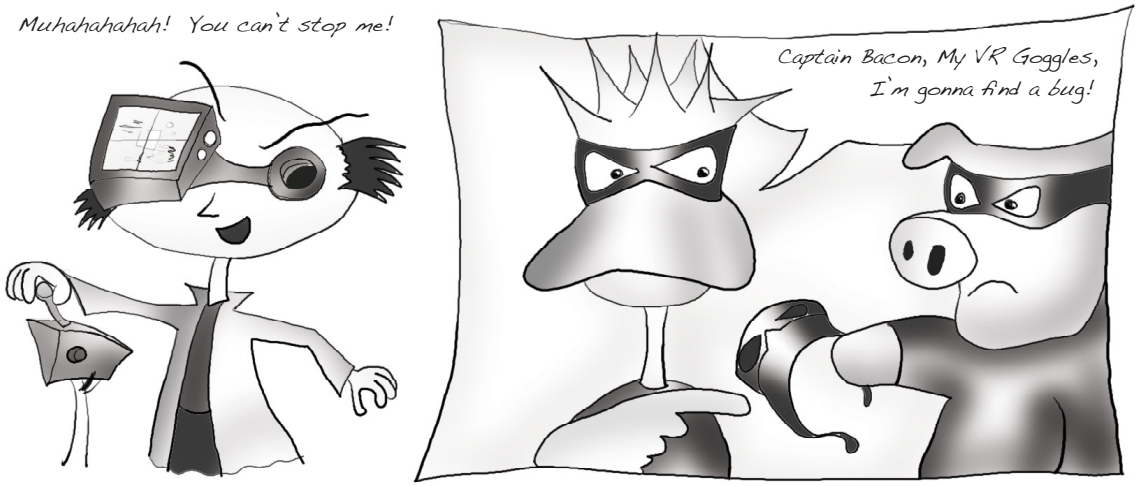


"It's simple to verify that a function throws when expected," says Dr. Surly, "Just wrap it in a Try block in your test, fail if an error wasn't thrown or if it's the wrong error."

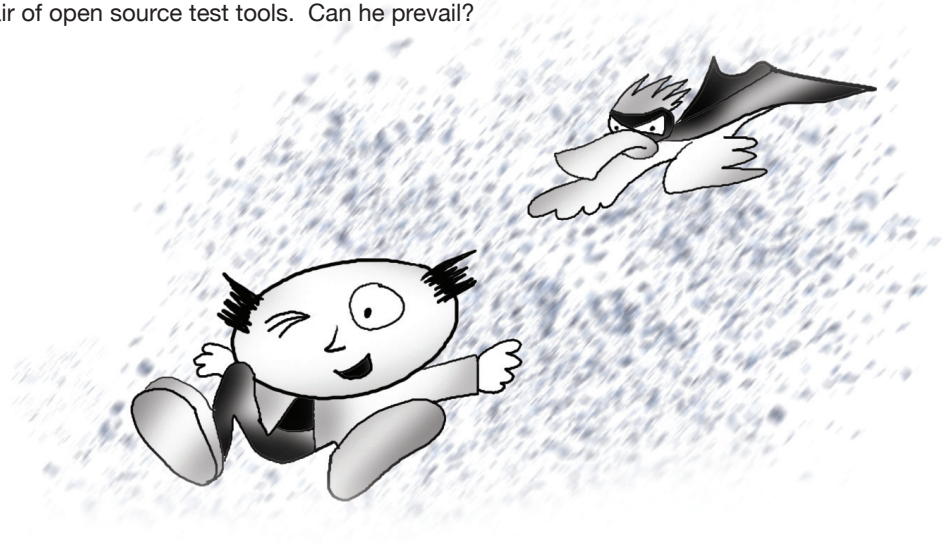
```
void test_VerifyThrow(void)
{
  CEXCEPTION_T e;
  Try
  {
    FunctionToTest();
    TEST_FAIL_MESSAGE(
      "no throw!");
  }
  Catch(e)
  {
    TEST_ASSERT_EQUAL(2,e);
  }
}
```

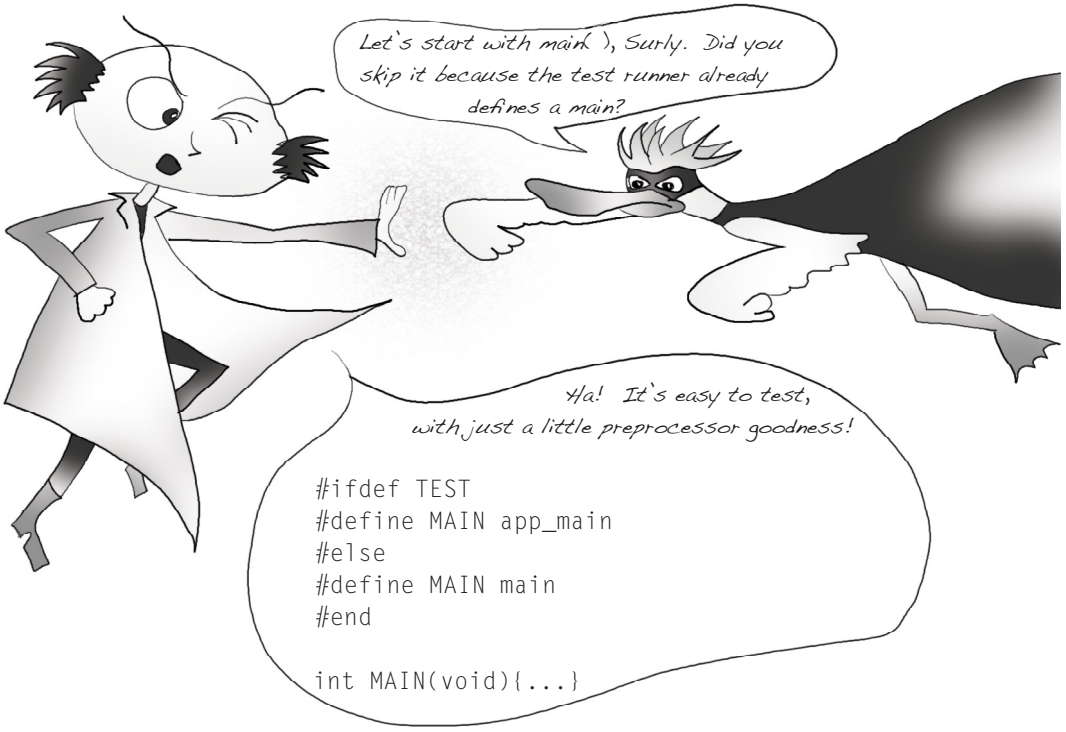
"... and," continues Dr. Surly, "It's easy to verify a function catches an error. The :cexception plugin adds ExpectAndThrow functions which can for a called function to throw an error at any desired time."

```
void test_VerifyCatch(void)
{
  CEXCEPTION_T e = 5;

  SubFunc_Expect(22);
  SubFunc2_Expect(33);
  SubFunc2_ExpectAndThrow(44, e);

  //Test Runner will catch
  //if uncaught here and will
  //cause test failure
  FunctionToTest();
}
```



Dr. Surly throws the switch. His masterful invention lumbers to life with an ominous hum. Thunder Duck knows his only option is to dive into the chaotic world of embedded software and find a bug... any loophole which he can exploit to intercept Dr. Surly's dastardly plans. But he's up against an evil genius, a sloth, and a pair of open source test tools. Can he prevail?





"It's a common convention to define TEST when compiling tests and not defining it when building a release. While we want our main source to contain as little code related to testing as possible, there are instances such as in main where a little hack is really useful."

"Like what else?" Thunderduck asks, momentarily distracted.

"Let's say you have a private function in your C file by declaring it static. If you just declare it STATIC instead, and use our new TEST friend to define STATIC to static during releases but nothing otherwise, we have an easy way to make private functions visible during tests. This can allow us to provide more rigorous tests on the guts of a module if we desire."

"Doesn't that make your tests more brittle?"

"A little," Dr Surly admits, "The tests are more closely linked to your implementation. It's a tradeoff, for sure."

"So what about an infinite loop?" Thunder Duck challenges, "Embedded systems often have infinite loops. As soon as you call that function in a test, it'll execute forever and you'll never get results!"

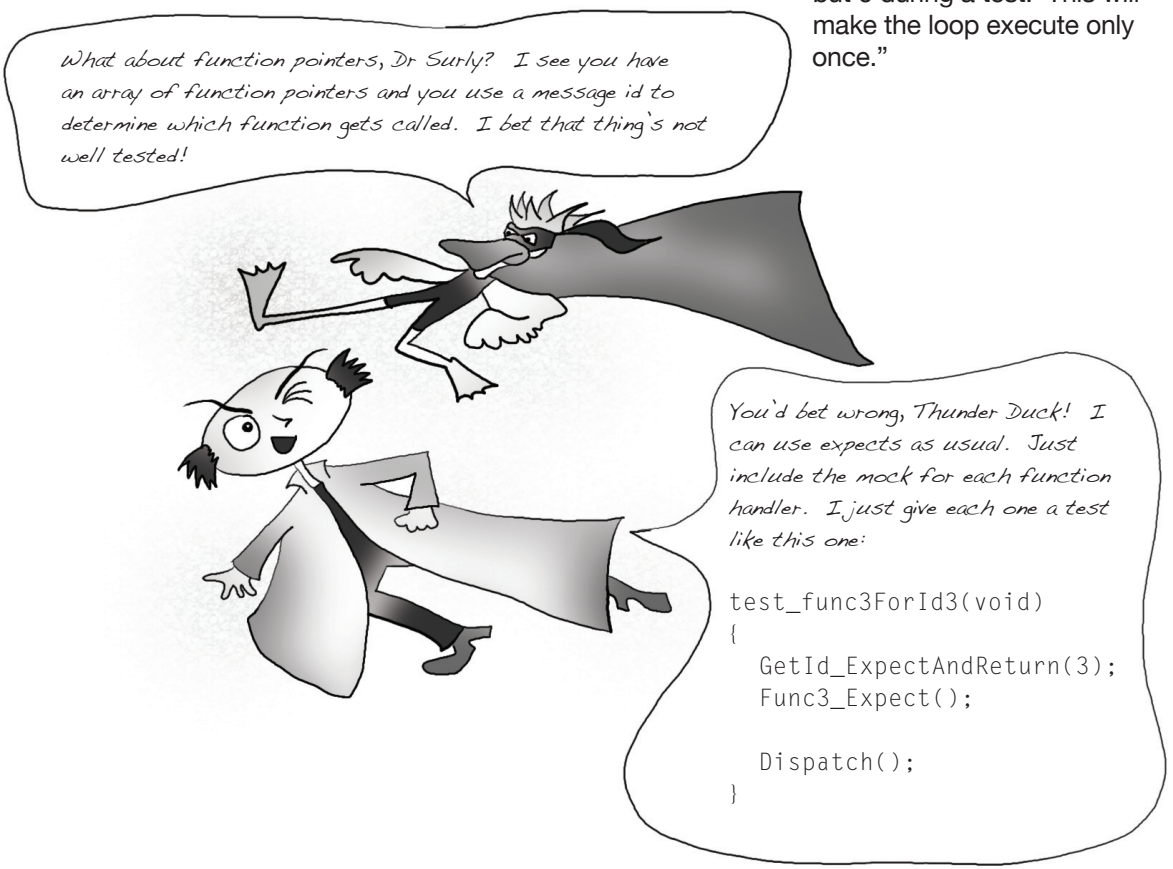
"Nonsense," Dr. Surly states, "Instead of using the usual options like these:"

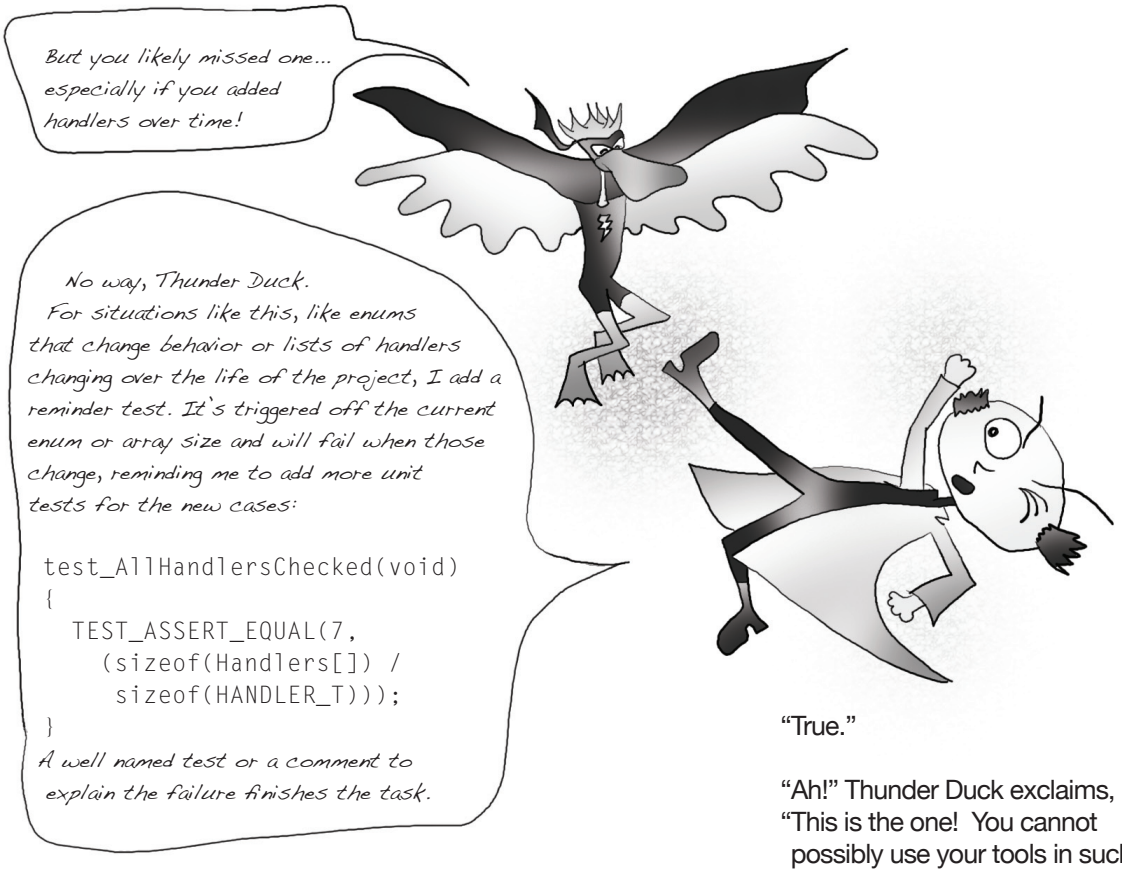
```
while(1) { ... }
for(;;) { ... }
```

"use a slightly more verbose method..."

```
do {...}
while(FOREVER);
```

"...where FOREVER is defined to be 1 during a release build but 0 during a test. This will make the loop execute only once."





"Ok, Surly," Thunder Duck smirks, "Let's say you have a large body of existing code... maybe an RTOS, a library, or a block of re-used legacy code... Surely you're not going to test it all."

"Not likely," he admits.

"Even assuming that the guts of this system work properly, how do you use your precious mocks? This thing could be a huge collection of header files, a maze of #ifdefs and #ifndefs, or might even include conflicting declarations depending on which files you link to!"

"True."

"Ah!" Thunder Duck exclaims, "This is the one! You cannot possibly use your tools in such a situation! You'll be stuck for sure!"

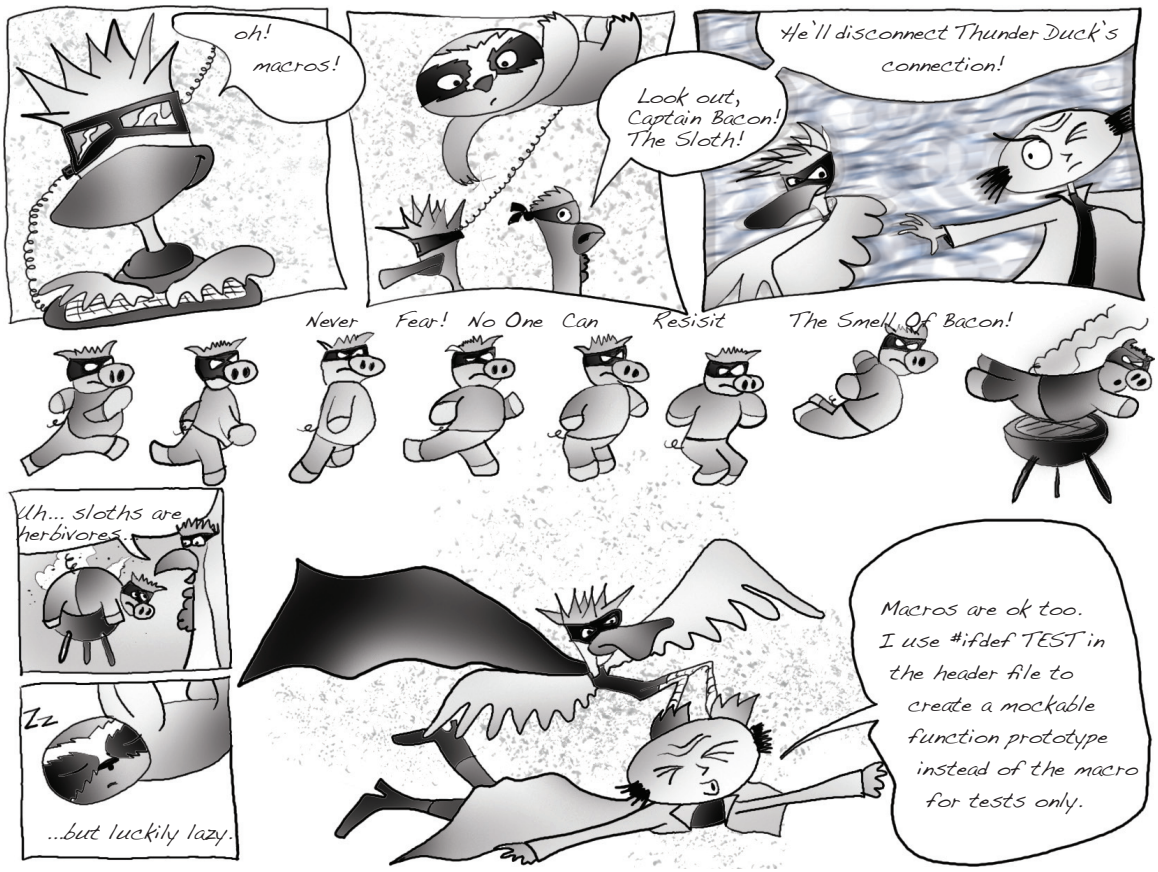
"It's fine," the mad scientist assures him, "It just takes a little bit of work. Instead of linking to all of those files, you want to create a single header which is the API you're using."

"Huh?"

"Let's say you're going to use FreeRTOS, a decent open source real-time operating system. Instead of including a header file for queues, mutexes, semaphores, tasks, and whatnot, you create a single header called something like FreeRTOS.h or maybe OS_API.h. In this file, you put function prototypes for all the functions that you are using in the OS. This is creating a great reference (documentation!) for how you are using the RTOS, as well as a single file that you can mock and use in any tests that make OS calls," Dr. Surly says with a satisfied smile.

"I suppose you'd just include the types too?"

"Most likely."



"Wait a minute!" says Thunder Duck, "So you are saying you would take macro and function definitions that are meant to be public, and collect them all in one header like this?"

```
int OS_TaskDef(FUNC*, int);
int OS_SemaSet(int);
int OS_SemaWait(int, int);

//Note: declared as macros elsewhere during release.
#ifdef TEST
int OS_Mutex_Enter(int);
int OS_Mutex_Exit(int);
#endif

//Note: defined here for convenience during release
#ifdef TEST
int OS_SemaSetFromISR(int);
#else
#define  OS_SemaSetFromISR(a) \
       { OS_SemaphoreSet(a); SWI; }
#endif
```

"Yes," Dr. Surly agrees, "That is exactly what I mean."

"I am sure that could get tedious if the library you're talking about has a lot of custom types or conditionals," insists Thunder Duck.

"Yes, it could," Dr. Surly says, "But you could always choose to run the library code through a preprocessor to generate that header if it was really bad... but honestly the manual method is usually not as bad as it seems."

"So that just leaves testing the library itself?"

"Sure, but most people will just write tests to verify their assumptions about the API of the library or a few simple tests... you leave the thorough testing to the library provider or it's long track record."

"Hurumph," Thunderduck grumbles, searching through more lines of code, "Ah! What about this place where you use the ignore plugin!? It looks like you needed to know how many times the function was called, but just didn't care about the actual arguments? But the ignore plugin doesn't work that way!"

"True, the ignore plugin defaults to behaving as if :ignore has been set to :args_and_calls... switching to :args_only will change the behavior of that plugin and make it care about how often functions are called, just like an _Expect, but not care about the actual arguments. In either case, the return values are handled the same."

"What if you're checking an argument's pointer address instead of the value?"

"I always have CMock in :smart pointer mode... I'm checking for NULL or dereferencing my pointers to check their contents. Otherwise I suppose you could use that TEST define again to make those local variables public during a test," Dr. Surly muses.

```
#ifdef TEST
int Num;
#endif
void FuncToBeTested(void)
{
  #ifndef TEST
  int Num;
  #endif
...
}
```

"I know!" Thunder Duck says, "Side effects! In C, people will often pass a pointer to a buffer that they expect to be filled by a function... or a pointer to some other type, but they expect the function being called to modify it in some way. How do you go about mocking something like that? In all your work so far, I have seen arguments checked for expected values, but only the return value is actually updated."

"For that," Dr. Surly says, smiling mischievously, "You need another of CMock's plugins. Just turn on the :callback plugin and you suddenly have a lot of power in how you can customize your tests."

"Explain," Thunder Duck growls, not liking the sound of this plugin.

"Let's say you have a function like this, which you knew would update val then return a status code."

```
int DoSomething(short* val);
```

"Instead of calling your _Expect's, you will call DoSomething_StubWithCallback once at the beginning of the test. You're mostly telling CMock that you have special handling for this function and that it should call your custom written stub function instead of doing the normal mock stuff. The stub has the same argument and return types as the function being stubbed."

"There are a couple of options that are important here. First, you can leave :callback_after_arg_check as false to just call your callback function, or true to check the arguments as an expect would before calling your callback function."

"There is also :callback_include_count which will add an additional argument to your callback function which is an integer containing the number of times the callback has been fired. This option is set to true by default."

"Let's say I've gone with the default options and each time I want the number argument to be set to the number of times the function has been called times the incoming value. Also, let's say we want to make sure our callback is only called twice."

```
int MyDoSomethingStub(short* val,int NumCalls) {
  *val = val * NumCalls;
  if (NumCalls >= 2)
    TEST_FAIL("called too many times");
  return 0;
}
```

```
void test_DoSomething_shouldActuallyDoIt(void) {
  DoSomething_StubWithCallback(MyDoSomethingStub);
  AnotherFunction_Expect(45);
  YetMore_ExpectAndReturn(5);

  //This calls DoSomething
  DoSomethingCaller(45, 3);
}
```

"This seems much more complicated than the expects and ignores," Thunder Duck laughs.

"True," Dr Surly says, "But the callback plugin gives you some real power. You can use it to perform complex calculations based on incoming data, to fill buffers or other things passed by reference, or even to write your own fancy expect calls which check only certain arguments. You're in full control of how the mocked out function will behave, a trade of power for complexity."

"Couldn't you accomplish the same thing by just writing a test stub for all your functions on your own?"

"Of course you could, but this is the best of both worlds... The stub interfaces are generated in addition to the Expect calls and whatever other plugins you have enabled. You can use automatically generated mocks most of the time, and just use the complicated version when absolutely necessary."

Thunder Duck scowls and resumes his search through Surly's code. Somewhere in here there must be a problem... something that cannot be caught!

Then he sees it. A slow smile grows across his face.
"You're testing in a simulator, right Surly?"

"Yes... I wanted to test using the same compiler that my release was built with," Dr. Surly says proudly.

"This is your linker file?" Thunder Duck asks, referencing a linker file that had been thrown together quickly for tests. Like most test linker files, it was quick and dirty... lots of extra memory for the tests... no real organization. Usually that's fine.

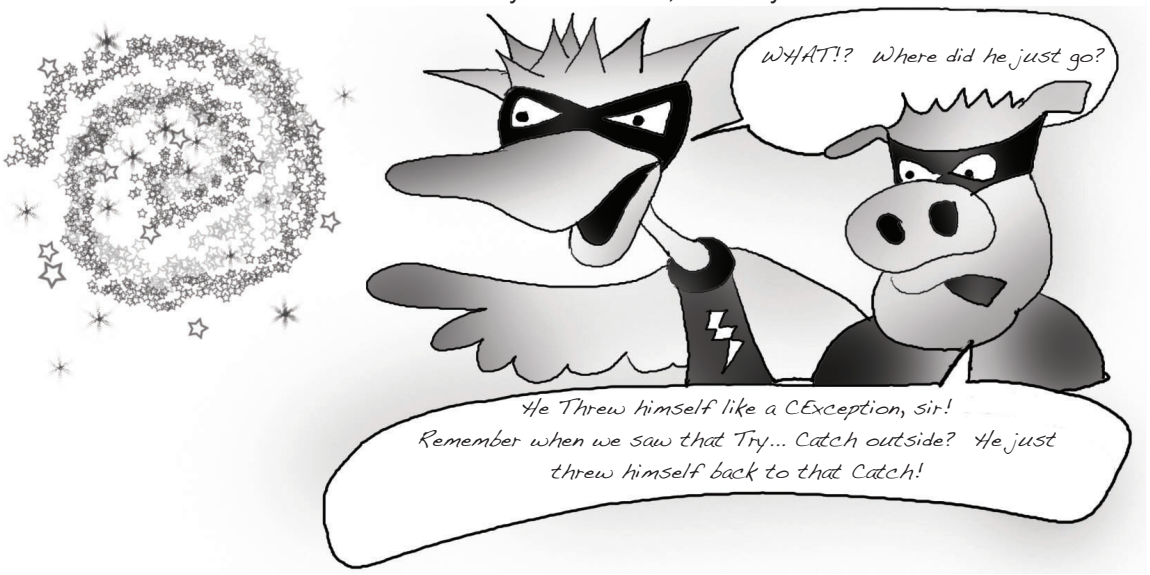
"Yes," Dr. Surly says, pausing. Something seems amiss.

The Duck smiles triumphantly,
"One of your test strings could get linked to start at address zero! You're not doing anything to map where any of the code or data goes... and if it DOES get linked to zero, any pointers to it are going to get interpreted as NULLS!"

"That seems highly unlikely," Dr. Surly says, but he's paled slightly... or he would be more pale if he wasn't so pasty white from all those years indoors.

“Unlikely or not... it’s possible!” Thunder Duck thunders. He scans through all the test cases quickly in search of a test that might have this fault. He feels anxious. The likelihood of this bug occurring is slim... but the linker file wasn’t set up to avoid such a mistake. It is possible!

But before Thunderduck can declare victory over his foe, Dr. Surly vanishes!



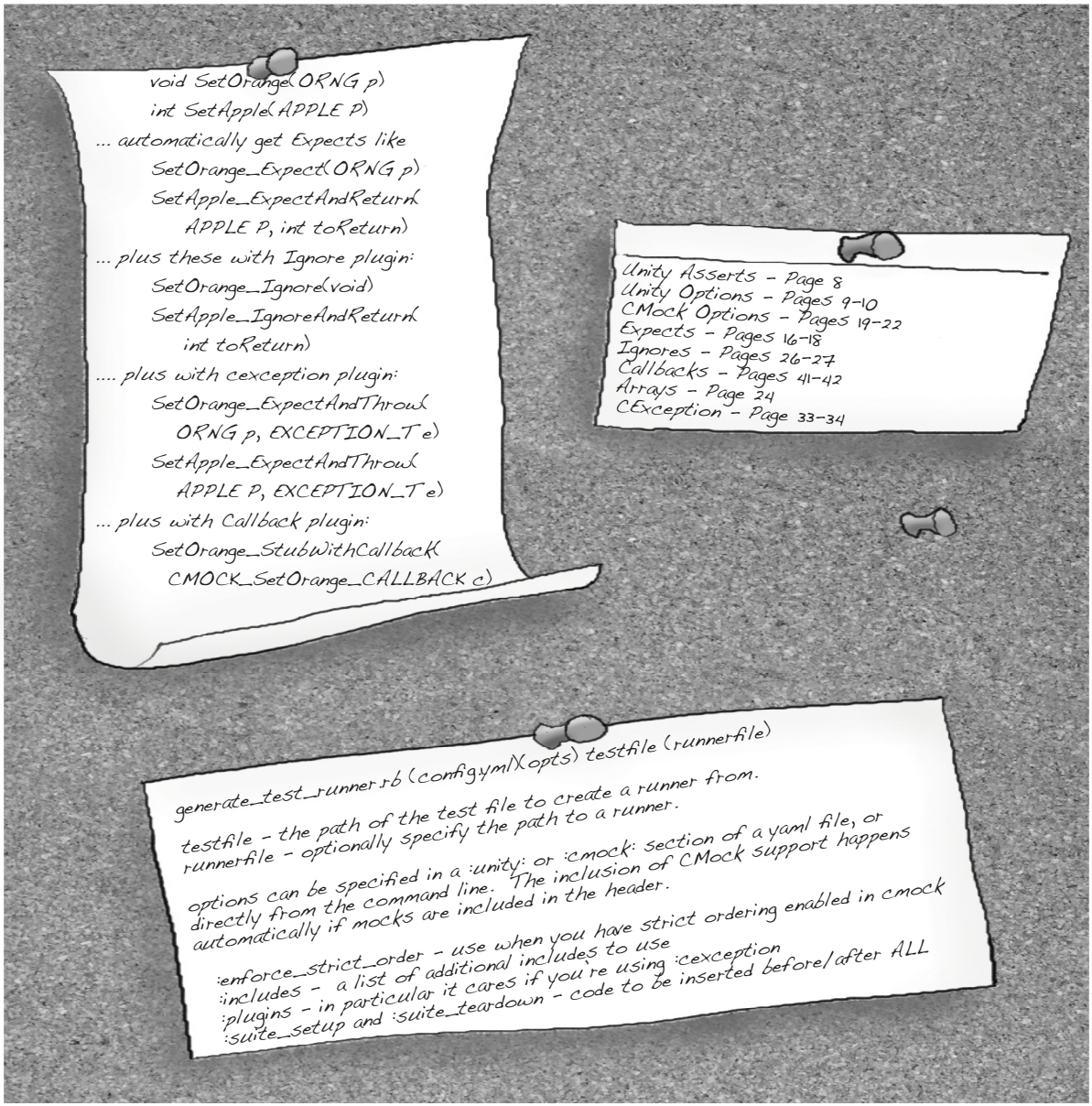
“The one page 33?” Thunder Duck demands, “When you asked if I thought I heard the Thunder Mobile?”

“Yes,” squeaks the pig.

As the realization settles in of just WHO stole the Thunder Mobile, Thunder Duck is filled with a rage!

And so we must now leave the Billed Crusader, The Mad Scientist, and the other misfits that fill these pages. While there are likely many unanswered questions remaining, that’s what forums and sequels are for.

For now... don’t you have some code of your own that you should be writing?


void SetOrange(ORNG p)
int SetApple(APPLE p)
... automatically get Expects like
SetOrange_Expect(ORNG p)
SetApple_ExpectAndReturn(
APPLE p, int toReturn)
... plus these with Ignore plugin:
SetOrange_Ignore(void)
SetApple_IgnoreAndReturn(
int toReturn)
.... plus with cexception plugin:
SetOrange_ExpectAndThrow(
ORNG p, EXCEPTION_T e)
SetApple_ExpectAndThrow(
APPLE P, EXCEPTION_T e)
... plus with Callback plugin:
SetOrange_StubWithCallback(
CMOCK_SetOrange_CALLBACK c)
Unity Asserts - Page 8
Unity Options - Pages 9-10
CMock Options - Pages 19-22
Expects - Pages 16-18
Ignores - Pages 26-27
Callbacks - Pages 41-42
Arrays - Page 24
CException - Page 33-34
generate_test_runner.rb (config.yml/X opts) testfile (runnerfile)
testfile - the path of the test file to create a runner from.
runnerfile - optionally specify the path to a runner.
options can be specified in a :unity: or :cmock: section of a yaml file, or
directly from the command line. The inclusion of CMock support happens
automatically if mocks are included in the header.
:enforce_strict_order - use when you have strict ordering enabled in cmock
:includes - a list of additional includes to use
:plugins - in particular it cares if you're using :cexception
:suite_setup and :suite_teardown - code to be inserted before/after ALL