Mike Long’s Blog

NOVEMBER 9, 2009 · 8:31 PM

Unit Testing C code with the GoogleTest framework

In a previous article I described how to get started testing C++ code with the Google Test framework. In this article I’m going to share a few tips and tricks for testing C code.

So what’s the big deal, isn’t it just the same as C++?

Well yes, in a way it is, but as always the devil is in the details. Here are some of the challenges we face when trying to test procedural code:

- We can’t create an instance of the code under test. This means that we can’t easily get a fresh object with initialized data for every test.
- The dependencies are hardcoded. This means we can’t use dependency injection techniques to mock/fake the module dependencies.
- We can’t use Polymorphism to break the dependencies

So that only leaves us with the two dependency-breaking tools available in the language: the preprocessor and the linker.

Things to watch out for

Static initialization: You need to be able to reset your data to a known state before running each test case. It is the only way to isolate your tests from each other.
Global variables: Is you module accessing a global variable? You need to provide a fake implementation for this.
Hardware Access: In embedded systems we often have memory mapped hardware register access. You most definitely don’t want to be dereferencing from random memory addresses in your tests. A good antidote to this is to define a generic function to get the address for a given register. You can then define a version of this function for testing purposes.

An example

So how does that look in practice? Suppose we have a make-believe embedded software application for controlling a device:

```c
#include <stdio.h>
#include <unistd.h>

#define IOMEM_BASE 0x2FF
#define VALUE_REG  (IOMEM_BASE + 3)

// This must be a power of 2!
#define BUFFER_SIZE 8
#define MAX_ITEMS (BUFFER_SIZE-1)
```

```
static int my_filter[BUFFER_SIZE];
static int readIdx = 0;
static int writeIdx = 0;

int filter_len(){ return (BUFFER_SIZE + writeIdx - readIdx) % BUFFER_SIZE; }

void filter_add(int val) {
 my_filter[writeIdx] = val;
 writeIdx = (writeIdx+1) & BUFFER_SIZE-1;
 if(writeIdx == readIdx) readIdx = (readIdx+1) & BUFFER_SIZE-1;
}

#ifndef TESTING
int myapp_do_dangerous_io()
{
 // lets dereference an io mapped register
 // - on the target it is at address IOMEM_BASE + 3
 return *((int *)VALUE_REG);
}
#endif

int myapp_get_average(){
 int len = filter_len();
 if(0 == len) return 0;
 int sum = 0;
 for(int i = 0; i < len; i++){
 sum += my_filter[(i+readIdx)%BUFFER_SIZE];
 }
 return sum/len;
}

int myapp_task()
{
 // get value from register
 int nextval = myapp_do_dangerous_io();

 // add to filter line
 filter_add(nextval);

 // return the average value as the next delay
 return myapp_get_average();
}

int myapp_mainloop()
{
 for(;;){
 int nextloopdelay = myapp_task();
 sleep(nextloopdelay);
 }
}

#ifndef TESTING
int main() {
 printf("!!!Hello World!!!\n");
 return myapp_mainloop();
}
#endif
```

How do we approach testing this nastyness?

There are some challenges to testing code of this nature, but there are also methods we can use overcome them.

Infinite loops: these guys will destroy your ability to test effectively. The best approach is to move the body of any infinite loop into it’s own function call.
Dangerous Code: what you do on hardware in production can be dangerous to do in a testing environment. In this example we have a hardware access of memory mapped IO address. There are three ways we can deal with dilemma:

1. change the address we dereference,
2. change the function we call (at link time)
3. hide the function we call during testing using #ifdefs and provide a test fake (this is the approach I have taken here)

Incompatible Function Names: You can't link two main functions. You need to hide one…

Static Memory: This can really hurt the independence of your tests. You really ought to re-initialize all of your static data for each test case, and thankfully there is an easy way to achieve this. All the major testing frameworks have a concept of a test fixture which allows you to call a SetUp function before execution of each test case. Use this to initialize your static data. Remember: independent tests are good tests!

The General Testing Pattern

1. Define fake functions for the dependencies you want to stub out
2. If the module depends on a global (gasp!) you need to define your fake one
3. include your module implementation (#include module.c)
4. Define a method to reset all the static data to a known state.
5. Define your tests

```
#include <gtest/gtest.h>

 // Hide main
 #define TESTING
 // Hide the io function since this will segfault in testing
 int fake_register;
 int myapp_do_dangerous_io()
 {
 return fake_register;
 }
 #include "myapp.c"

 class MyAppTestSuite : public testing::Test
 {
 void SetUp(){
 memset(&my_filter, 0, sizeof(my_filter));
 readIdx = 0;
 writeIdx = 0;
 }

 void TearDown(){}
 };

 TEST_F(MyAppTestSuite, myapp_task_should_return_correct_delay_for_one_element) {
 fake_register = 10;
 EXPECT_EQ(10, myapp_task());
 }

 TEST_F(MyAppTestSuite, myapp_task_should_return_correct_delay_for_two_elements) {
 fake_register = 10;
 myapp_task();
 fake_register = 20;
 EXPECT_EQ(15, myapp_task());
 }

 TEST_F(MyAppTestSuite, get_average_should_return_zero_on_empty_filter) {
 ASSERT_EQ(0, myapp_get_average());
 }

 TEST_F(MyAppTestSuite, addFirstFilterValAddsVal) {
 filter_add(42);
 ASSERT_EQ(42, my_filter[readIdx]);
 }

 TEST_F(MyAppTestSuite, addFirstReturnsCorrectAverage) {
 filter_add(42);
 ASSERT_EQ(42, myapp_get_average());
 }

 TEST_F(MyAppTestSuite, addTwoValuesReturnsCorrectAverage) {
 filter_add(42);
 filter_add(40);
 ASSERT_EQ(41, myapp_get_average());
 }

 TEST_F(MyAppTestSuite, get_average_should_return_average_of_full_filter) {
 for(int i = 0; i < MAX_ITEMS; i++){
 filter_add(i);
 }
 ASSERT_EQ((0+1+2+3+4+5+6)/MAX_ITEMS, myapp_get_average());
 }

 TEST_F(MyAppTestSuite, get_average_should_return_average_of_wrapped_filter) {
 for(int i = 0; i < BUFFER_SIZE; i++){
 filter_add(i);
 }
 ASSERT_EQ((1+2+3+4+5+6+7)/MAX_ITEMS, myapp_get_average());
 }

 /// ....test buffer operations...
 ...
```

That's all well and good, but what about <difficult thing>?

When talking about testing C code (especially embedded) I often hear "But what about.."

- Timing Problems. That's right, unit testing can't magically simulate the runtime properties of your system.
- Interrupts. This is a special case of the last point, but it is the same issue all developers come across when going multi-threaded.
- Bit-correct operations. If you are running 24-bit code on a 32-bit architecture you will not see the exact same behavior for various overflow, underflow, bit-shifting and arithmetic operations.
- I can't possibly test this! Well, there are some classes of code that simply cannot be tested using the unit testing methodology. In my experience however, it is an extreme minority of most code bases that this applies to. The secret is to factor out impossible-to-test-code as much as possible so you don't pollute the rest of the codebase.

Summary

Testing C code is hard. Testing legacy C code is even harder. But with the limited dependency-breaking language features we have in C (the linker and the preprocessor) we can accomplish quite a lot.

You can view the original source code on GitHub.
————————————————

With apologies to Michael Feathers…

- BEYOND: Two Souls and Tribeca http://bit.ly/18vBc4n
-
-
-
- http://bit.ly/18vBc4n

15 Responses to *Unit Testing C code with the GoogleTest framework*

Andrew

August 3, 2010 at 1:48 am

Do you think googletest is suitable for testing C, or do you think there are more appropriate ones? I know googletest certainly can be used to create unit tests for C, but I don't know if the extra work to jerry it is worth it, compared to one designed originally for C.

Reply

meekrosoft

August 3, 2010 at 9:48 am

I think it really depends on the situation. If you have a project that is completely C then it would probably be a better idea to use a framework written in C. On the other hand, if you have a project with both C and C++

googletest might be best.

For me, I would go for googletest anyway because I really value some of the extra features it comes with (for instance it is really easy to integrate with the hudson CI server, and the ability to choose which tests are run at runtime).

Reply

Pingback: Unit Testing Legacy C | Mike Long's Blog

Pingback: Diverse läsning | La Vie est Belle

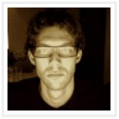
Michael
January 3, 2012 at 5:40 pm

My limited experience is that unit testing C code not designed to be tested is a walk in the park compared to it's C++ equivalent. I work on rather low-level mixed C/C++ software. With the C parts it is normally feasible to isolate a reasonable subset of procedures and break their dependencies with linker and pre-processor as you say. With massive C++ classes with many dependencies (including two COM variants) I find the added dependency-breaking tools available fall far short of compensating for the additional pain.

Re testing timing and interrupt issues, I have nasty thoughts involving unit tests designed to run inside valgrind's helgrind tool.

Reply

meekrosoft
January 18, 2012 at 12:48 am



Thanks for your thoughts Michael!

Reply

Daniel Umaña
January 17, 2012 at 5:07 pm

Hi Mike! I have a question that maybe you can help me with…

We just recently are starting a testing project for embedded C code, and we are in the phase were we have to select a testing framework. We are really looking to use googletest as our framework, but we have encountered an issue which maybe you can suggest a solution, or already have seen the problem and can help us, or maybe just point me to any link that can suggest a starter.

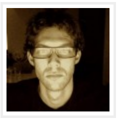
Here it is,

We have separate modules, some of them requires using mocks. We know
we can make the mocks manually, but we were thinking to use some sort
of usefull tool provided for google test. I´ve read about google
mocks, but unfortunately they are developed in C++, and we cant use it
if we are developing and testing pure C code – it would require to
change a lot of stuff on the original code, which for testing purposes
we are not allowed to touch.

So, my question is, does someone already have encountered this
problem? Is there any mock solution that google test provide for C
pure? Does anybody has some link that I can read regarding this?

Every help would be much appreciated, dont hesitate to ask for more
information if it is needed in order for you to give a better help on
this topic.

Reply



meekrosoft
January 18, 2012 at 12:48 am

Hi Daniel,
In my last project we used a simple framework to fake free functions, you can find out more here:
https://github.com/meekrosoft/fff

It is easy to use, just download a header file into your project and you can create mocks with one line of code. At the moment it doesn't handle const parameters but I am working on supporting it now.

I know others have used a framework called CMock which I believe is good but we didn't use it because it needs to run before the build to parse your header files and generate the mock code. IT also requires ruby. If this isn't an issue for you then it might be worth evaluating also.

Let me know if you have any more questions!

Reply

Daniel Umaña
January 20, 2012 at 7:45 pm

Thank you for your quick anser. You are very kind!

so, while reading what you posted, I need to ask for some clarification. These might be some dumb questions, but I need to understand a little bit…

1) fff is a mock framework that is compatible with google test? I see you are not using gtest on the fff examples you provide for your project. Or am I not seeing something?

2) If we decide to use CMock, do you know anyone who has already made the union between gtest and Cmock happen? Is it easy possible, or we have to travel to a very difficult path?

Thanks for your answer, and for this blog!

-Daniel

Mike Long

January 24, 2012 at 1:20 pm

Hi Daniel,

Yes, fff is compatible with any test framework, including google test. Actually the test suite for fff is written using googletest, you can view some examples here:
https://github.com/meekrosoft/fff/blob/master/test/fff_test_cpp.cpp

As for the second question, I'm not sure I can answer that because I don't have any experience there. My instinct would say that it is compatible but you best check with the project to be sure.

Hope that helps!

Michael

September 7, 2012 at 3:29 pm

Hi Mike! Great Blog! I've got a question to the example in this article.
Is there a reason for including the module implementation (myapp.c) into the testcode? Isn't it more practicable to compile this module separately and link both objects (test.o and myapp.o) together? What are the Pro's and Con's?

Reply

meekrosoft

September 8, 2012 at 5:34 am

Hi Michael, thanks for the feedback! The reason to #include the implementation file into the test file is that it allows you to use the preprocessor to remove difficult dependencies (in this case the myapp_do_dangerous_io() function). This is an example of a preprocessor seam. I have written a post on test seams that you might find helpful:
http://meekrosoft.wordpress.com/2012/07/20/test-seams-in-c-function-pointers-vs-preprocessor-hash-defines-vs-link-time-substitution/

Reply

sharath

January 9, 2013 at 3:34 am

Hi I am currently looking out for some good unit test framework for Embedded C programs. I have a doubt in Using google C++ test framework. I want to know whether can we test the inter OS communication methods using this framework like Message queues between two threads or between two process in embedded board? or is it basically limited to unit testing only?

Reply

meekrosoft

January 13, 2013 at 12:30 pm

It really depends on your situation, but googletest is designed as a unit testing framework. I have used it in the previous projects for Hardware in the loop testing, but always the test suite would run off target. For testing in target I think that a developing a custom testing strategy is the only choice.

Reply

Pingback: [c/c++] google testをカレントディレクトリ配下のみで利用する | 情報基盤システム学研究室