Mike Long's Blog

OCTOBER 4, 2009 · 5:19 PM

Testing C++ code with the GoogleTest framework

There are some great surveys of the C++ testing frameworks out there, my favorite being the one at Games From Within. It was based on that article that my project started using the CxxTest framework when we got started with automated testing. CxxTest really is a great testing framework, and very well suited to embedded systems development. It requires only a limited subset of the C++ language so that makes it very portable and requires very little from the compiler.

Soon however I became feature-hungry. First I wanted to be able to report my test coverage in our Hudson CI server. Secondly, I became interested in reducing the scaffolding required to develop tests. Thirdly, because CxxTest is a perl script that generates a cpp file from your hxx test file, it increased the complexity of the build rules in the project.

The first objective could have been quite easily achieved if I were to develop an output writer for CxxTest that would output as JUnit xml, but together with the other objectives it seemed like a good time to try the GoogleTest framework.

GETTING STARTED WITH GOOGLETEST

Writing your first test with GoogleTest couldn't be simpler. Simply include the code you want to test, include the google mock framework, and write your tests.

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

TEST(MathTest, TwoPlusTwoEqualsFour) {
  EXPECT_EQ(2 + 2, 4);
}
```

Of course this is the most basic test you can write, but this ease of use significantly reduces the friction required to get started. The most interesting thing to note from the code snippit is the complete lack of scaffolding code required. There is no need to create test classes or to specifically register your tests, this is done automagically from the googletest framework. Of course as you write more and more tests you will start to notice repetitious setup and teardown code, and that's where fixture classes can help.

FIXTURES

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

using namespace std;

// A new one of these is created for each test
class VectorTest : public testing::Test
{
public:
  vector<int> m_vector;

  virtual void SetUp()
```

```
13      {
14        m_vector.push_back(1);
15        m_vector.push_back(2);
16      }
17
18      virtual void TearDown()
19      {
20      }
21  };
22
23  TEST_F(VectorTest, testElementZeroIsOne)
24  {
25    EXPECT_EQ(m_vector[0], 1);
26  }
27
28  TEST_F(VectorTest, testElementOneIsTwo)
29  {
30    EXPECT_EQ(m_vector[1], 2);
31  }
32
33  TEST_F(VectorTest, testSizeIsTwo)
34  {
35    EXPECT_EQ(m_vector.size(), (unsigned int)2);
36  }
```

There are a few interesting points to note here:

- Firstly, a new instance of the fixture class is created for every test associated with the fixture. This is important because it isolates each test from the side effects of other tests.
- Secondly, the test declaration has changed from TEST to TEST_F. This lets the framework know that it shouldn't generate the default fixture class but instead look for one you defined.
- Finally, it knows which fixture class to use by matching the first test parameter (here it is VectorTest). This allows you to have any number of fixture classes and tests in your code.

So what are the runtime steps for each test you define?

1. Instantiate a fixture class
2. Call SetUp on the fixture
3. Call the test
4. Call TearDown on the fixture
5. Destroy the fixture

When you run the test suite you should see an output like the following:

```
[mlong@n6-ws2 x86]$ bin/hellotest
Running main() from gtest_main.cc
[==========] Running 4 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from VectorTest
[ RUN      ] VectorTest.testElementZeroIsOne
[       OK ] VectorTest.testElementZeroIsOne (0 ms)
[ RUN      ] VectorTest.testElementOneIsTwo
[       OK ] VectorTest.testElementOneIsTwo (0 ms)
[ RUN      ] VectorTest.testSizeIsTwo
```

```
[       OK ] VectorTest.testSizeIsTwo (0 ms)
[----------] 3 tests from VectorTest (0 ms total)

[----------] 1 test from MathTest
[ RUN      ] MathTest.Zero
[       OK ] MathTest.Zero (0 ms)
[----------] 1 test from MathTest (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 2 test cases ran. (0 ms total)
[  PASSED  ] 4 tests.
[mlong@n6-ws2 x86]$
```

You can also specify to run only a subset of tests at the command line, either by name or by using wildcards.

PARAMETERIZED TESTS

Sometimes your tests can become repetitive because you want to test your code with many different inputs. The google test framework helps you by allowing you to specify value and type parameterized test suites. This allows you to run your tests multiple times with either different data types or different values. This can be a huge productivity boost when testing code for multiple inputs. Here is a test suite that performs all the tests for three different input values, “meek”, “geek”, and “freek”. Individual tests can then access the value for which the test is being run by using the GetParam() function call.

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

using namespace std;

// Function under test...
bool acceptName(const char* name)
{
  char *result = strstr(name, "eek");
  if(result == NULL){
    return false;
  }
  return true;
}

// A new one of these is create for each test
class MyStringTest : public testing::TestWithParam<const char*>
{
public:
  virtual void SetUp(){}
  virtual void TearDown(){}
};

INSTANTIATE_TEST_CASE_P(InstantiationName,
                        MyStringTest,
                        ::testing::Values("meek", "geek", "freek"));

TEST_P(MyStringTest, acceptsEekyWords)
{
  ASSERT_TRUE(acceptName(GetParam()));
}
```

In addition to value-parameterized tests you can also create type-parameterized tests. Check out the project wiki for more information.

THE END

So far we have been very pleased with the GoogleTest framework. The features more than meet the original objectives we had and more. We use the framework to test at all levels in our system, from 8 bit microcontroller code, 24 bit dsp code, embedded linux processes, integration level, and even full hardware system integration testing. If you're interested to find out more, go check out the GoogleTest project wiki. In addition, there is a great C++ mocking library called GoogleMock which complements GoogleTest, and I'll talk more about this in a later article.

5 Responses to *Testing C++ code with the GoogleTest framework*

Dana

February 19, 2011 at 2:40 am

I am enjoying your blog quite a bit – thank you!

I'm interested in running GoogleTest on an ARM target. Were your experiences with the framework on the host or the target? Can you comment on the difficulty/ease of porting GoogleTest to an embedded target?

Reply

meekrosoft

February 19, 2011 at 3:36 pm

Hi Dana,
Thanks for dropping by!

Running the tests on the target will depend mostly on the target capabilities. To run googletest you will need an operating system to perform the I/O, so if you are running an embedded linux you should be ok. Googletest is available in source so it should be no problem to compile for your target if you have a relatively modern compiler.

If you wish to use googlemock in addition beware that there are more requirements (specifically the tr1/tuple library) which can be harder to get running if you have an older gcc.

Running the tests on the host is a breeze, assuming of course you have written your software in a hardware independent manner.

Hope this helps!

Mike

Reply

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

Avneet Kaur

April 9, 2013 at 5:50 pm

How to run these examples?
I am using this
g++ -I${GTEST_DIR}/include -I${GTEST_DIR} -c ${GTEST_DIR}/src/gtest-all.cc ar -rv libgtest.a gtest-all.o

But getting errors
libgtest.a: No such file or directory

Reply

meekrosoft

April 9, 2013 at 9:52 pm

Hi Avneet,
What is your development operating system? What version of gcc are you using?

Reply