N4420 - Defining Test Code

Robert Douglas and Michael Price
April 8th, 2015

Introduction

Testing has gone beyond the purview of the responsible programmer, and become an integral part of the development process.

Tests are a means to aid in design, discovery of functionality, documentation of intent, explanation of behavior, protection from regressions, and easing maintenance burdens. The standard, however, provides no means by which a developer can define their test code and share it in a manner that does not pollute their production binaries.

As of this time, the state-of-the-art for writing tests in C++ comes down to the authors choice in a third party tool, almost certainly built entirely on macros, and most often forcing the most basic tests to be located distant to the code to which they refer. From some anecdotes, build systems may even introduce additional dedicated libraries to just facilitate testing, so as to prevent pollution of the production binary.

Existing Practice

Each language has varying levels of support for testing, and highly varied designs for writing and executing tests. The following is a small sampling from a variety of languages, in the means they employ to enable testing. For links to these, please see the references section at the end of this paper.

C++

C++ sports many available libraries, including Boost Test, CppUnit, CUTE, Google Test, and Qt Test. None of these are standardized or shipping with the language, and each diverges in features and usage patterns. The large open field of frameworks and libraries also leads to many proprietary solutions.

Python

Doctest and unittest modules provide ability to write tests in comments and/or in test classes.

Java

The JUnit and TestNG libraries provide annotations (roughly equivalent to C++ attributes) and support various conventions that allow test code to be identified and which can be used by tools to execute and report results.

D

For D, the tests can be defined with the code-under-test. This is accomplished by the keyword ‘unittest.’ Given the proper switch, the D Compiler will compile test code into the executable.

The binary that is produced will execute the tests after static initialization and before entry into the “main” function.

Proposal

To distinguish ‘test code’ from ‘non-test code’, we propose to add a new qualifier or attribute. For the purpose of this paper, we use the placeholder, ‘test’:

```
void foo() test
{
    struct Inner {}; // implied test
}

struct MyFixture test
{
    void memberFn(); // implied test
    struct Inner {}; // implied test
};

template<typename T>
struct MyTemplate test {};

template<typename T>
void templateFn() test {}

int main() test {}
```

Code marked "test" (test-code) can only be referenced by other test-code. 'Test' is a qualifier of a function or class. Only a 'test' function can create an instance of a test class, directly call a test function, take reference to a test function, or make reference of a test class as a type passed to a template. A test-class implicitly has all member functions and sub-classes marked 'test', and likewise for non-member functions.

```
struct TestClass test {};

struct NonTestClass {};

void nonTestFn();

void testFn() test {
    TestClass tc; // Ok
    NonTestClass ntc; // Ok
    nonTestFn(); // Ok
    testFn(); // Ok
    std::function<void()> ntf(&nonTestFn); // Ok
    std::function<void()> tf(&testFn); // Ok
}

void nonTestFn() {
    nonTestFn(); // Illegal. testFn is not visible to non-test code
    std::function<void()> fn(&testFn); // Illegal. testFn is not visible to non-test code
    TestClass tc; // Illegal. TestClass is not visible to non-test code
}
```

The relationship between test-code and non-test code, is similar to the relationship between mutable and const. That is, test-code can be given by test-code to non-test code. Notice, above, the passing of `nonTestFn` to `std::function<void()>`.

With this distinction of test-code and non-test code, it becomes possible to define an additional version of main, qualified by 'test.' The upshot of this feature, allows us to define an entry-point to the program, to run the tests. By separating out the tests from the non-tests, the production binary can benefit from omitting the test code, generating a smaller footprint, while still benefiting from the compile-time checks the test-code performs. It is expected, then, that a well built compiler can then emit a second binary with the test-main as the entry point, allowing the application developer to define how they exercise their test code.

Example: (to compile this example, simply add `#define test`, to the top of the file)

```
#include <algorithm>

template<typename BiDirectionalIterator>
bool isPalindrome(BiDirectionalIterator first, BiDirectionalIterator last)
{
    if (std::distance(first, last) > 1)
    {
        for (; first < last; ++first, --last)
        {
            if (*first != *last) return false;
        }
    }
    return true;
}

#include <string>
#include <stdexcept>

void isPalindrome_assertTrue_ForEmpty() test
{
    std::string emptyStr;
    if (!isPalindrome(emptyStr.begin(), emptyStr.end()))
        throw std::runtime_error("empty string should be palindrome");
}
void isPalindrome_assertTrue_For1Elem() test
{
    std::string str("1");
    if (!isPalindrome(str.begin(), str.end() - 1))
        throw std::runtime_error("1 element string should be palindrome");
}
void isPalindrome_assertTrue_ForManyElems_evenCount() test
{
    std::string str("123321");
    if (!isPalindrome(str.begin(), str.end() - 1))
        throw std::runtime_error("Even number of elements failed");
}
void isPalindrome_assertTrue_ForManyEems_oddCount() test
{
    std::string str("12321");
    if (!isPalindrome(str.begin(), str.end() - 1))
```

```
        throw std::runtime_error("Odd number of elements failed");
}
void isPalindrome_assertFalse_ForManyElems_evenCount() test
{
    std::string str("123456");
    if (isPalindrome(str.begin(), str.end() - 1))
        throw std::runtime_error("Even number of elements failed");
}
void isPalindrome_assertFalse_ForManyEems_oddCount() test
{
    std::string str("12345");
    if (isPalindrome(str.begin(), str.end() - 1))
        throw std::runtime_error("Odd number of elements failed");
}

#include <iostream>
#include <vector>
#include <functional>

int main() test
{
    std::vector<std::function<void()>> tests {
        &isPalindrome_assertTrue_ForEmpty,
        &isPalindrome_assertTrue_For1Elem,
        &isPalindrome_assertTrue_ForManyElems_evenCount,
        &isPalindrome_assertTrue_ForManyEems_oddCount,
        &isPalindrome_assertFalse_ForManyElems_evenCount,
        &isPalindrome_assertFalse_ForManyEems_oddCount
    };
    std::for_each(tests.begin(), tests.end(), [](std::function<void()> const& fn){
        try
        {
            fn();
            std::cout << "Test Passed" << std::endl;
        }
        catch (std::runtime_error e)
        {
           std::cerr << "Failed: " << e.what() << std::endl;
        }
    });
}
```

When the tests can accompany the code, the behavior is defined and proven.

Not Included in this Proposal

1. Bikeshedding of keyword/attribute
2. Assertion library
3. Explicit way to reflect over all the defined test functions
4. Prescribed means for running all the tests
5. Distinction between types of tests: unit/acceptance/integration/examples/etc

Requested Straw Polls

- We like the distinction between test and non-test code
    - We like the ability to define a test-main

- We like the direction of keyword-based qualification
- We like the direction of an attribute-base qualification
- We want to be able to define a separate test-main
- We want this, even without accompanying assertion library
- We want this, even without reflection-like test-discovery
- We would like to see an exploration of namespace test-attribution

Acknowledgements

Thanks to Alex Kondratskiy for his help in this domain. Also, thanks to KCG for their continued support in this endeavour.

References

"25.2. Doctest — Test Interactive Python Examples." *25.2. Doctest*. Web. 08 Apr. 2015. <https://docs.python.org/2/library/doctest.html>.

"25.3. Unittest — Unit Testing Framework." *25.3. Unittest*. Web. 08 Apr. 2015. <https://docs.python.org/2/library/unittest.html>.

"Boost Test Library." *Boost Test Library*. Web. 08 Apr. 2015. <http://www.boost.org/doc/libs/1_57_0/libs/test/doc/html/index.html>.

"CUTE." *Writing and Running Unit Test Suites*. Web. 08 Apr. 2015. <http://cute-test.com/projects/cute/wiki/Writing_and_Running_CUTE_Unit_Test_Suites>.

"CppUnit Cookbook." *CppUnit*. Web. 08 Apr. 2015. <http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html>.

"JUnit - About." *JUnit - About*. Web. 08 Apr. 2015. <http://junit.org/>.

"Qt Test 5.4." *Qt Test 5.4*. Web. 08 Apr. 2015. <http://doc.qt.io/qt-5/qttest-index.html>.

"TestNG - Welcome." *TestNG - Welcome*. Web. 08 Apr. 2015. <http://testng.org/doc/index.html>.

"Unit Tests." - *D Programming Language*. Web. 08 Apr. 2015. <http://dlang.org/unittest.html>.

"Googletest - Google C Testing Framework - Google Project Hosting." *Googletest - Google C Testing Framework - Google Project Hosting*. Web. 08 Apr. 2015. <https://code.google.com/p/googletest/>.