Google Unit Test Framework

victor

January 21, 2009

1. Tests should be *independent* and *repeatable*. It’s a pain to debug a test that succeeds or fails as a result of other tests. Google C++ Testing Framework isolates the tests by running each of them on a different object. When a test fails, Google C++ Testing Framework allows you to run it in isolation for quick debugging.

2. Tests should be well *organized* and reflect the structure of the tested code. Google C++ Testing Framework groups related tests into test cases that can share data and subroutines. This common pattern is easy to recognize and makes tests easy to maintain. Such consistency is especially helpful when people switch projects and start to work on a new code base.

3. Tests should be *portable* and *reusable*. The open-source community has a lot of code that is platform-neutral, its tests should also be platform-neutral. Google C++ Testing Framework works on different OSes, with different compilers (gcc, MSVC, and others), with or without exceptions, so Google C++ Testing Framework tests can easily work with a variety of configurations. (Note that the current release only contains build scripts for Linux - we are actively working on scripts for other platforms.)

4. When tests fail, they should provide as much *information* about the problem as possible. Google C++ Testing Framework doesn’t stop at the first test failure. Instead, it only stops the current test and continues with the next. You can also set up tests that report non-fatal failures after which the current test continues. Thus, you can detect and fix multiple bugs in a single run-edit-compile cycle.
5. The testing framework should liberate test writers from housekeeping chores and let them focus on the test content. Google C++ Testing Framework automatically keeps track of all tests defined, and doesn’t require the user to enumerate them in order to run them.

6. Tests should be fast. With Google C++ Testing Framework, you can reuse shared resources across tests and pay for the set-up/tear-down only once, without making tests depend on each other.

```cpp
#include
#include

using namespace std;

// Returns n! (the factorial of n). For negative n, n! is defined to be 1.
int Factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

// Returns true iff n is a prime number.
bool IsPrime(int n) {
    // Trivial case 1: small numbers
    if (n <= 1) return false;

    // Trivial case 2: even numbers
    if (n % 2 == 0) return n == 2;

    // Now, we have that n is odd and n >= 3.
    // Try to divide n by every odd number i, starting from 3
```
for (int i = 3; ; i += 2) {
    // We only have to try i up to the square root of n
    if (i > n/i) break;

    // Now, we have i <= n/i < n.
    // If n is divisible by i, n is not prime.
    if (n % i == 0) return false;
}

// n has no integer factor in the range (1, n), and thus is prime.
return true;
}

// Step 2. Use the TEST macro to define your tests.

// TEST has two parameters: the test case name and the test name.
// After using the macro, you should define your test logic between a
// pair of braces. You can use a bunch of macros to indicate the
// success or failure of a test. EXPECT_TRUE and EXPECT_EQ are
// examples of such macros. For a complete list, see gtest.h.

// In Google Test, tests are grouped into test cases. This is how we
// keep test code organized. You should put logically related tests
// into the same test case.

// The test case name and the test name should both be valid C++
// identifiers. And you should not use underscore (_) in the names.

// Google Test guarantees that each test you define is run exactly
// once, but it makes no guarantee on the order the tests are
// executed. Therefore, you should write your tests in such a way
// that their results don’t depend on their order.

// Tests Factorial().

// Tests factorial of negative numbers.
TEST(FactorialTest, Negative) {
    // This test is named ‘Negative’, and belongs to the ‘FactorialTest’
// test case.
EXPECT_EQ(1, Factorial(-5));
EXPECT_EQ(1, Factorial(-1));
EXPECT_TRUE(Factorial(-10) >= 0);

//

// EXPECT_EQ(expected, actual) is the same as
// EXPECT_TRUE((expected) == (actual))
// except that it will print both the expected value and the actual
// value when the assertion fails. This is very helpful for
// debugging. Therefore in this case EXPECT_EQ is preferred.
//
// On the other hand, EXPECT_TRUE accepts any Boolean expression,
// and is thus more general.

// Tests factorial of 0.
TEST(FactorialTest, Zero) {
  EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, Positive) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}

// Tests IsPrime()

// Tests negative input.
TEST(IsPrimeTest, Negative) {
  // This test belongs to the IsPrimeTest test case.
  EXPECT_FALSE(IsPrime(-1));
  EXPECT_FALSE(IsPrime(-2));
EXPECT_FALSE(IsPrime(INT_MIN));

// Tests some trivial cases.
TEST(IsPrimeTest, Trivial) {
    EXPECT_FALSE(IsPrime(0));
    EXPECT_FALSE(IsPrime(1));
    EXPECT_TRUE(IsPrime(2));
    EXPECT_TRUE(IsPrime(3));
}

// Tests positive input.
TEST(IsPrimeTest, Positive) {
    EXPECT_FALSE(IsPrime(4));
    EXPECT_TRUE(IsPrime(5));
    EXPECT_FALSE(IsPrime(6));
    EXPECT_TRUE(IsPrime(23));
}

int main(int argc, char** argv) {
    // Prints elapsed time by default.
    // testing::GTEST_FLAG(print_time) = true;

    // This allows the user to override the flag on the command line.
    testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

Google test
Unit Test
Running 6 tests from 2 test cases.

1 Global test environment set-up.

1 3 tests from FactorialTest

1 FactorialTest.Negative

1 FactorialTest.Negative

1 FactorialTest.Negative

1 FactorialTest.Zero

1 FactorialTest.Zero

1 FactorialTest.Positive

1 FactorialTest.Positive

1 3 tests from isPrimeTest

1 isPrimeTest.Negative

1 isPrimeTest.Negative

1 isPrimeTest.Trivial

1 isPrimeTest.Trivial

1 isPrimeTest.Positive

1 isPrimeTest.Positive

1 Global test environment tear-down

1 6 tests from 2 test cases ran.

PASSED 1 6 tests.

Process returned 0 (0x0) execution time : 0.070 s

Press any key to continue.