Unit test using gtest

Tuesday, April 21, 2020 12:21 PM

Hello World example:

Test.cpp

```
#include <iostream>
#include <gtest/gtest.h>
using namespace std;
int main(int argc, char ** argv){
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Compile using:

```
g++ Test.cpp -lgtest -lgtest_main -pthread
```

Add test by:

```
TEST(TestName, Subtest_1) {
    ASSERT_TRUE(1==1);
}
```

Assertions:

Success, Non-Fatal Failure, Fatal Failure

EXPECT_EQ() ASSERT_EQ()

Fatal Failure means when assertion fails, the code exists from the test.

Equal : EXPECT_EQ(), ASSERT_EQ()

Not equal: EXPECT_NE(), ASSERT_NE().

Less than: EXPECT_LT(), ASSERT_LT()

Less than equal: EXPECT_LE(), ASSERT_LE()

Greater than: EXPECT_GT(), ASSERT_GT().

Greater than equal: EXPECT_GE(), ASSERT_GE().

Non Fatal Failure Fatal Failure

We should only have one assertion in each test.

Test

Arrange, Act, Assert

```
TEST(TestName, increment_by_5){
    // Arrange
    int value = 100;
    int increment = 5;
    // Act
    value = value + increment;
    // Assert
    ASSERT_EQ(value, 105);
}
```

Unit test should run fast, within ms.

must be able to run independently.

doesn't depend on external input.

```
class MyClass{
    string id;
public:
    MyClass(string _id) : id(_id) {}
    string GetId() { return id;}
};
TEST(TestName, subtest) {
    // Arrange
    Myclass mc("root");
    // Act
    string value = mc.GetId();
    // Assert
    ASSERT_EQ(value.c_str(), "root");
}
```

↑ This should be ASSERT_STREQ.

For strings, use

equal: EXPCT_STREQ(), ASSERT_STREQ()

non fatal failure fatal failure

Test Fixture

```
#include <iostream>
#include <gtest/gtest.h>
using namespace std;
class MyClass {
    int baseValue;
public:
    MyClass(int _bv): baseValue(_bv) {}
    void Increment(int byValue){
        baseValue += byValue;
    }
    int getValue() { return baseValue;}
};
TEST(ClassTest, Increment_by_5) {
    // Arrange
    MyClass mc(100);
    // Act
    mc.Increment(5);
    // Assert
    ASSERT_EQ(mc.getValue(), 105);
}
int main(int argc, char **argv){
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

We can use test fixture for the common arrange part:

```
struct MyClassTest : public testing::test {
    MyClass *mc;
    void SetUp() {mc = new Myclass(100);}
    void TearDown() {delete mc;}
}
```

We can change the test function to use test fixture:

```
TEST_F(MyClassTest, Increment_by_5){
    // Act
    mc->Increment(5);
    // Assert
    ASSERT_EQ(mc->getValue(), 105);
}
```

The struct is called once in every test.

```
#include <iostream>
#include <vector>
#include <gtest/gtest.h>
using namespace std;
class Stack{
    vector<int> vstack = {};
public:
    void push(int value) { vstack.push_back(value);}
    void pop() {
        if(vstack.size() >0){
            int value = vstack.back();
            vstack.pop_back();
            return value;
        } else {
            return -1
        }
    }
    int size() {return vstack.size();}
};
struct stackTest : public testing::Test{
    Stack s1;
    void SetUp() {
        int value[] = {1,2,3,4,5,6,7,8,9};
        for(auto &val : value) {
            s1.push(val);
        }
    }
    void TearDown() {}
};
TEST_F(stackTest, PopTest){
    int lastPoppedValue = 9;
    while(lastPoppedValue != 1)
        ASSERT_EQ(s1.pop(), lastPoppedValue --);
}
int main(int argc, char ** argv){
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

We can add another test

```
TEST_F(stackTest, sizeValidityTest){
    int val = s1.size();
    for (val; val >0; val--)
        ASSERT_NE(s1.pop(), -1);
}
```