

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:

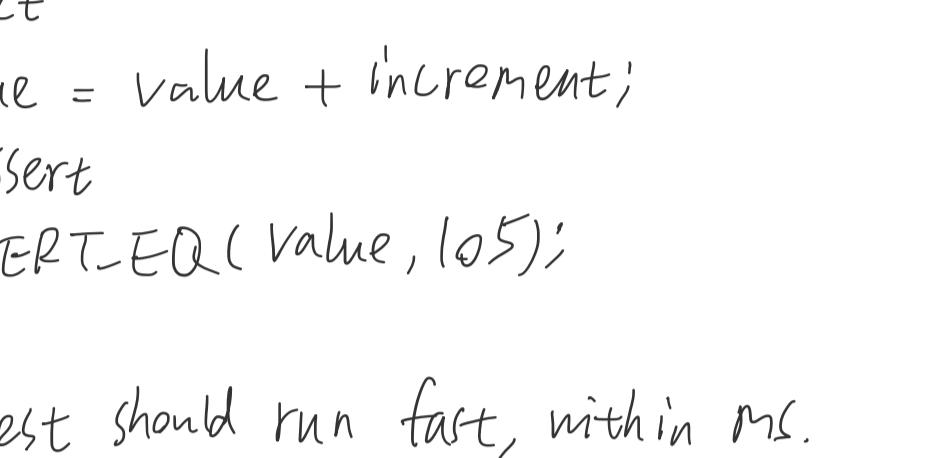
```
g++ Test.cpp -lgtest -lgtest_main -lpthread
```

Add test by:

```
TEST(TestName, subtest_1) {
    ASSERT_TRUE(l == 1);
```

Assertions:

Success, Non-fatal failure, Fatal failure

 EXPECT_EQ() ASSERT_EQ()

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

Equal : EXPECT_EQ(), ASSERT_EQ()

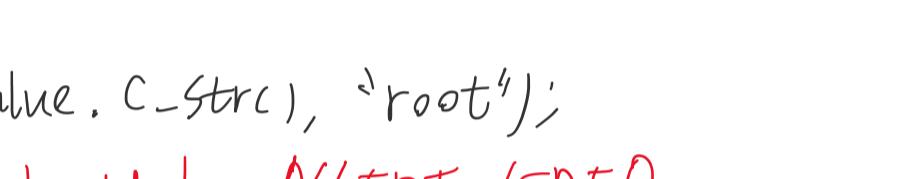
Not equal: EXPECT_NE(), ASSERT_NE()

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 = 10;

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: EXPECT_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 bvValue) {
```

baseValue += bvValue;

}

int getValue() { return baseValue; }

}

```
TEST(TestName, increment_by_5) {
```

// Arrange

MyClass mc(10);

// 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(10); }
```

```
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);

}

```
int main(int argc, char **argv) {
```

testing::InitGoogleTest(&argc, argv);

return RUN_ALL_TESTS();

}

We can add another test

```
TEST_F(MyClassTest, sizeValidityTest) {
```

int val = mc->size();

```
for (int i = 0; i < val; i++) {
```

mc->push(i);

}

```
int size() { return mc->size(); }
```

}

```
void TearDown() {
```

}

```
TEST_F(MyClassTest, PopTest) {
```

int lastPoppedValue = 9;

while (lastPoppedValue != 1) {

ASSERT_EQ(mc->pop(), lastPoppedValue - 1);

}

```
int main(int argc, char **argv) {
```

testing::InitGoogleTest(&argc, argv);

return RUN_ALL_TESTS();

}

We can add another test

```
TEST_F(MyClassTest, sizeValidityTest) {
```

int val = mc->size();

```
for (int i = 0; i < val; i++) {
```

mc->push(i);

}

```
int size() { return mc->size(); }
```

}

```
void TearDown() {
```

}

```
TEST_F(MyClassTest, PopTest) {
```

int lastPoppedValue = 9;

while (lastPoppedValue != 1) {

ASSERT_EQ(mc->pop(), lastPoppedValue - 1);

}

```
int main(int argc, char **argv) {
```

testing::InitGoogleTest(&argc, argv);

return RUN_ALL_TESTS();

}