Lab Work 11 :Unit Testing Using Google Testing Framework

Create a project in Visual C++ using google Test

Creating Tests

Except for a main function at the end, the remainder of the file consists of tests.

To create a test: use the TEST macro to define and name a test function. Each test is an ordinary C++ functions that does not return a value.

```
TEST(test_case_name, test_name) {
…
test body
…
}
```

The two parameters comprise the name of the test. The Google Testing Framework uses the following terminology, which may be different from what you have learned:

•A test represents the execution of a single unit test. The test either passes or fails.

•A test case contains one or many tests.

The first parameter is the more general test case name. The second parameter is the more specific test name. The full name of the test consists of both the test case and the test name. The Google Testing Framework groups the test results by test cases, so logically-related tests should be in the same test case.

As an example, we will create two test cases for the function fib. Both of these tests are in the same test case called FibTest but the names of the individual tests reflect what they are testing.

```
// Tests fib of 0.
TEST(FibTest, HandlesZeroInput) {
…
}
// Tests fib of positive numbers. TEST(FibTest, HandlesPositiveInput) {
…
}
```

A couple of rules regarding test names:

•Each test must have a different full name. This means, for any two tests, either the test case name and/or the test name must be different.

•The test name can only contain letters and numbers.   Symbols, including underscores, are not permitted.

The test body consists of normal C++ syntax. A typical test will run the function under test at least once and compare the result to the expected result. The success or failure of a test is determined using assertions. An assertion is a statement that will check whether a condition is true. If the condition is true, the check is successful. If the condition is false, the check is unsuccessful or is a failure.

The Google Testing Framework consists of several assertion macros. We’ll focus on one right now (the others are described later in this document):

```
EXPECT_EQ(expected, actual);
```

The expected value represents the value that is expected and is typically calculated by hand. The actual value represents the value of the function under test. Here is an example for the fibonacci test HandlesZeroInput:

```
// Tests 0-th fib number. TEST(FibTest, HandlesZeroInput) {

EXPECT_EQ(0, fib(0));

}
```

In this test, the expected value is 0 and the actual value is the result returned from fib(0). A test may contain multiple assertions as seen here:

```
// Tests fib of positive numbers. TEST(FibTest, HandlePositiveInput) {

EXPECT_EQ(1, fib(1)); EXPECT_EQ(1, fib(2)); EXPECT_EQ(2, fib(3)); EXPECT_EQ(144, fib(12));

}
```

If one or more assertions in a test fail, the whole test is considered to fail. If none of the assertions fail (all of the assertions pass), then the test will pass.

Executing the Tests

Once all of the tests have been written, the end of the file needs to contain the following main function:

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

::testing::InitGoogleTest(&argc, argv);

return RUN_ALL_TESTS();

}
```

Executing the tests consists of creating a test program in the compiler and running the executable the compiler creates.

The compiler command is:

g++ fibtest.cpp fib.cpp -lgtest -lpthread -o fibtest

This command compiles the fibtest.cpp and fib.cpp files and creates an executable called fibtest. The switch –lgtest directs the compiler to use the Google Testing Framework library. The switch –lpthread directs the compiler to use the Pthread library, a library required by the Google Testing Framework.

To run the tests, simply execute the executable fibtest:

./fibtest

You should see output like this:

```
[cs1]$./fibtest
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from FibTest
[ RUN      ] FibTest.HandleZeroInput
[       OK ] FibTest.HandleZeroInput (0 ms)
[ RUN      ] FibTest.HandlePositiveInput
[       OK ] FibTest.HandlePositiveInput (0 ms)
[----------] 2 tests from FibTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 2 tests.
```

In this situation both of the tests pass.

Let’s see what happens when a test fails. Edit some line of fib() in fib.cpp to make it misbehave (wrong calculation of Fibonacci numbers):

Recompile and re-execute fibtest. Now you should get output different from above:

A program may only have one function named main. You may have noticed that the normal function main for this program is in the file driver.cpp and not in fib.cpp. If the function main appeared in the file fib.cpp, the compiler would have returned an error stating there are two functions with the name main. When the test program was compiled, notice how the file main.cpp was absent. As a result,

there is only one function main in fibtest.cpp. If we wanted to create the actual executable, we would only compile fib.cpp and driver.cpp as specified in Makefile:

To summarize:

•To run the tests, a separate test program is created.

•The test program uses its own main function.

•The file containing code under test cannot have its own function main. It may be necessary to create a separate file that contains main.

Testing Classes

This section describes how to test classes using the Google Testing Framework.   In many ways, testing the member functions of a class is identical to that of testing any other function. The largest difference is the use of data members. Data members can be part of the input that needs to be set before the function runs. In addition, modifications to data members need to be checked to determine whether the test passes or fails. If modifications to data members are made incorrectly, the test should fail.

Since data members are private, there is an extra complication of setting and getting the data members. The testing functions are normal C++ functions and do not have direct access to data members. In most cases, we will rely on set and get functions during testing.

As an example,   copy the files Fraction.cpp, Fraction.h, and FractionTest.cpp.   The file

Fraction.h defines a fraction class and Fraction.cpp implements its member functions.

The file FractionTest.cpp contains tests for the class. Here is a test for one of the constructors:

```
TEST(FractionTest, Constructor) {
  Fraction a(3, 8);
  EXPECT_EQ(3, a.getNumer());
  EXPECT_EQ(8, a.getDenom());

  Fraction b(14, 6);
  EXPECT_EQ(14, a.getNumer());
  EXPECT_EQ(6, a.getDenom());
}
```

One thing you will notice is that the get member functions getNumer and getDenom are used in the assertion to check that the constructor properly set the values. Since fractions have two data members, most of the checks will require two assertions – one for the numerator and one for the denominator.

A few other comments regarding the tests for fraction:

•Many member functions are very simple (such as both constructors). As a result, only one or two tests are needed to test these functions. However, if error checking is part of a function, more tests will be needed to make sure the error checking is working correctly.

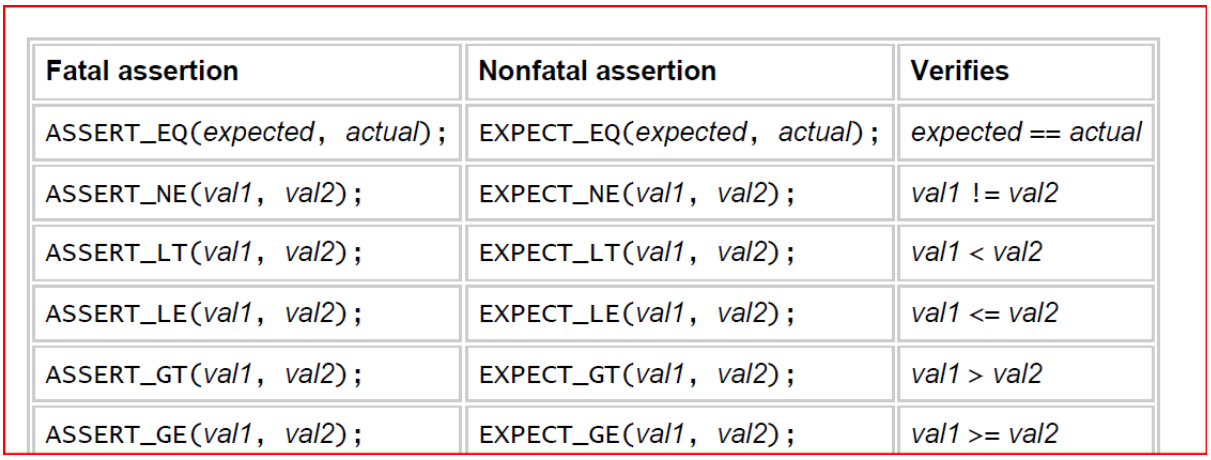
•The function display displays the fraction on the screen and nothing else (no data member are modified and no value is returned). It is not possible to write an assertion that checks to see if the screen contents are correct. Therefore, no unit tests are written for this member function. Instead, manual testing is needed to functions such as display and to check screen output in general.

Assertions (optional)

This section describes additional assertions that are part of the Google Testing Framework.

The assertions come in pairs that test the same thing but have different effects on the current function. ASSERT_* versions generate fatal failures when they fail, and abort the current function. EXPECT_* versions generate nonfatal failures, which don't abort the current function. Usually EXPECT_* assertions are preferred, as they allow more than one failures to be reported in a test. However, you should use ASSERT_* if it doesn't make sense to continue when the assertion in question fails.

Comparison-based assertions



Fatal assertion	**Nonfatal assertion**	**Verifies**
ASSERT_EQ(*expected*, *actual*);	EXPECT_EQ(*expected*, *actual*);	*expected* == *actual*
ASSERT_NE(*val1*, *val2*);	EXPECT_NE(*val1*, *val2*);	*val1* != *val2*
ASSERT_LT(*val1*, *val2*);	EXPECT_LT(*val1*, *val2*);	*val1* < *val2*
ASSERT_LE(*val1*, *val2*);	EXPECT_LE(*val1*, *val2*);	*val1* <= *val2*
ASSERT_GT(*val1*, *val2*);	EXPECT_GT(*val1*, *val2*);	*val1* > *val2*
ASSERT_GE(*val1*, *val2*);	EXPECT_GE(*val1*, *val2*);	*val1* >= *val2*

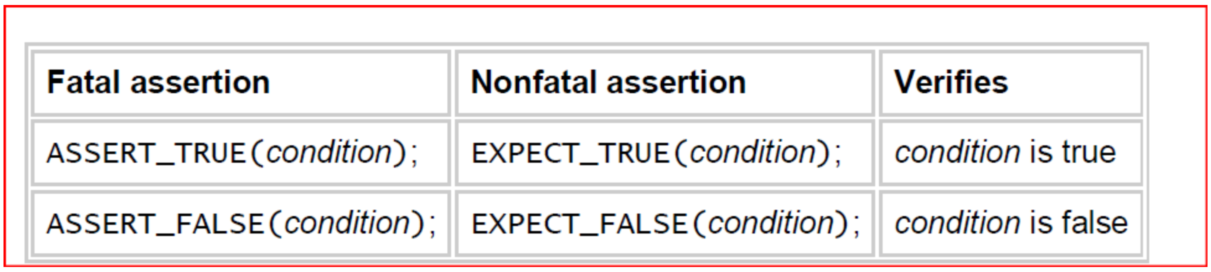
Some notes:

•These comparisons work with basic data types, strings, and any classes that have the corresponding operator overloaded.

•For pointers, it compares the memory addresses (not the contents of the pointers). Two pointers are equal if they point to the exact same memory location.

Boolean assertions

These assertions do basic true/false condition testing.



Fatal assertion	Nonfatal assertion	Verifies
ASSERT_TRUE(*condition*);	EXPECT_TRUE(*condition*);	*condition* is true
ASSERT_FALSE(*condition*);	EXPECT_FALSE(*condition*);	*condition* is false

String assertions

The assertions in this group compare two C-style strings (character arrays). If you want to compare two C++ string objects, use the comparison assertions such as EXPECT_EQ instead.

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_STREQ(*expected_str*, *actual_str*);	EXPECT_STREQ(*expected_str*, *actual_str*);	the two C strings have the same content
ASSERT_STRNE(*str1*, *str2*);	EXPECT_STRNE(*str1*, *str2*);	the two C strings have different content
ASSERT_STRCASEEQ(*expected_str*, *actual_str*);	EXPECT_STRCASEEQ(*expected_str*, *actual_str*);	the two C strings have the same content, ignoring case
ASSERT_STRCASENE(*str1*, *str2*);	EXPECT_STRCASENE(*str1*, *str2*);	the two C strings have different content, ignoring case

Note that a NULL pointer and an empty string are considered to be different.

Custom Failure Messages

To improve readability when an error occurs, you can add failure messages to assertions using the <<

operators such at this:

ASSERT_EQ(a.getDenom(), b.getDenom()) << "Fractions a and b have different denominators";

This can be helpful for comparing arrays:

```
for (int i = 0; i < x.size(); ++i) {
EXPECT_EQ(x[i], y[i]) << "Arrays x and y differ at index " << i;
}
```

Exercise 1

```
EXPECT_TRUE(2 + 2 == 2 * 2);
EXPECT_FALSE(1 == 2);
ASSERT_TRUE(2 + 2 == 2 * 2);
ASSERT_FALSE(1 == 2);
```

try all these on a empty google test project and see the outcome

Exercise 2

1. Prepare all these codes accordingly

```
// fibonacci.h
        #ifndef Fibonacci
        #define Fibonacci
        #include <iostream>
        using namespace std;
        int fibonacci(int number);
        #endif

        // fibonacci.cpp
        #include <iostream>
        #include "fibonacci.h"
        using namespace std;
        int fibonacci(int number){
          if(number <= 2){
            return 1;
          } else {
            return fibonacci(number - 1) + fibonacci(number - 2);
          }
        }
```

2. Write your test codes in test.cpp

```
#include <gtest/gtest.h>
            #include "../fibonacci.h"
            using namespace std;
            TEST(TestFibonacci, TestFunctionResults) {
              ASSERT_TRUE(fibonacci(1) == 1);
              ASSERT_TRUE(fibonacci(10) == 55);
              ASSERT_TRUE(fibonacci(16) == 987);
              ASSERT_TRUE(fibonacci(19) == 4181);
            }



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

              testing::InitGoogleTest(&argc, argv);

              return RUN_ALL_TESTS();
            }
```

Exercise 3

```
auto a = 42, b = 10;
EXPECT_EQ(a, 42);
EXPECT_NE(a, b);
EXPECT_LT(b, a);
EXPECT_LE(b, 11);
EXPECT_GT(a, b);
EXPECT_GE(b, 10);
```

Exercise 4

```
bool is_positive(int const val)
```

```
{
  return val != 0;
}
bool is_double(int const val1, int const val2)
{
  return val2 + val2 == val1;
}
TEST(TestAssertions, Predicates)
{
  EXPECT_PRED1(is_positive, 42);
  EXPECT_PRED2(is_double, 42, 21);

  ASSERT_PRED1(is_positive, 42);
  ASSERT_PRED2(is_double, 42, 21);
}
```

Additional Information

For more information on unit testing using the Google Testing Framework:

•Getting started with Google C++ Testing Framework (http://code.google.com/p/googletest/wiki/Primer)

•A quick introduction to the Google C++ Testing Framework (http://www.ibm.com/developerworks/aix/library/au-googletestingframework.html)