Googletest

Jan 25, 2021

Thinkings on test

Googletest

googletest helps you to write better *C++* tests.

Good tests

- Tests should be independent and repeatable.
- Tests should be well organized and reflect the structure of the tested code.
- Tests should be portable and reusable. (*platform-neutral*)
- Tests should be fast.
- When tests fail, they should provide as much information about the problem as possible.

Basic Concepts

- When using googletest, we start by writing *assertions*.
- An assertion's result can be **success**, **nonfatal failure**, **fatal failure**, if a fatal failure occurs, it aborts the current function; otherwise, the program continues normally.
- A *test suite* contains one or many tests. You should group your tests into test suites that reflect the sturcture of the code. When multiple tests in a test suite need to share common objects and subroutines, you can put them into a test fixture class.
- A test program can contain multiple test suites.

Assertions

When an assertion fails, googletest prints the assertion's source file and line number location, along with a failure message.

- `ASSERT_*` versions generate fatal failures when they fail and abort current function.
- `EXPECT_*` versions geenrate nonfatal failures, which don't abort current function.
- To provide a custom failure message, simply stream it to the macro using the << operator.

Basic Assertion

Fatal Assertion	Nonfatal Assertion	Verifies
`ASSERT_TRUE(condition)`	`EXPECT_TRUE(condition)`	`condition` is true
`ASSERT_FALSE(condition)`	`EXPECT_FALSE(condition)`	`condition` is false

Binary Comparison

Fatal Assertion	Nonfatal Assertion	Verifies
`ASSERT_EQ(val1, val2)`	`EXPECT_EQ(val1, val2)`	`val1 == val2`
`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`

- `ASSERT_EQ()` does pointer equality on pointers. If used on two C strings, it tests if they are in the same memory location, not if they have the same value.
- `ASSERT_EQ(actual, expected)` is preferred to `ASSERT_TRUE(actual == expected)`, since it tells you actual and expected's values on failure.
- The arguments'evaluation order is undefined.

String Comparison

The assertions in this group compare two **C strings**.
If you want to compare two **string objects**, use `EXPECT_EQ`, `EXPECT_NE`, and etc instead.

Fatal Assertion	Nonfatal Assertion	Verifies
`ASSERT_STREQ(val1, val2)`	`EXPECT_STREQ(val1, val2)`	The two strings have the same content
`ASSERT_STRNE(val1, val2)`	`EXPECT_STRNE(val1, val2)`	The two strings have different contents
`ASSERT_STRCASEEQ(val1, val2)`	`EXPECT_STRCASEEQ(val1, val2)`	The two strings have the same content, ignoring case
`ASSERT_STRCASENE(val1, val2)`	`EXPECT_STRCASENE(val1, val2)`	The two strings have different contents, ignoring case

- Note that "CASE" in an assertion name means that case is ignored.
- A NULL pointer and an empty string are considered different.

Test

- Use the `TEST()` macro to define and name a test function.
- In this function, along with any valid *C++* statements you want to include, use the various googletest assertions to check the values.
- The tests' results is determined by the assertions; if any assertions in the test fails, or if the tetst crashes, the entire test fails.
- `TEST()` arguments go from general to specific.
- A test's full name consists of its containing tests suite and its individual name.
- googletest groups the test results by test suites.

Test Fixtures

To reuse the same configuration of objects for several different tests.

- Derive a class from `::testing::Test`, start its body with `protected:`.
- Inside the class, declare any objects you plan to use.

- If necessary, write a default constructor or `SetUp()` function, to prepare the objects for each test.
- If necessray, write a default destructor or `TearDown()` function to release any resources allocated in `SetUp()`.
- When using a fixture, use `TEST_F()` instead, as it allows you to access objects in the fixture.

```
TEST_F(TestFixtureName, TestName){
    // test body
}
```

Like `TEST()`, the first argument is the test suite name, but for `TEST_F()` this must be the name of the test fixture class.

Invoking the Tests

`TEST()` and `TEST_F()` implicitly register their tests with googletest.
Most users should not need to write their own main function and instead link with `gtest_main` (as opposed to with `gtest`), which defines a suitable entry point.
The following is an example of fixture implementation:

```
#include "this/package/foo.h"
#include "gtest/gtest.h"

namespace my {
namespace project {
namespace {

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {
 protected:
  // You can remove any or all of the following functions if their bodies would
  // be empty.

  FooTest() {
     // You can do set-up work for each test here.
  }

  ~FooTest() override {
     // You can do clean-up work that doesn't throw exceptions here.
  }

  // If the constructor and destructor are not enough for setting up
  // and cleaning up each test, you can define the following methods:

  void SetUp() override {
     // Code here will be called immediately after the constructor (right
     // before each test).
  }

  void TearDown() override {
     // Code here will be called immediately after each test (right
     // before the destructor).
  }

  // Class members declared here can be used by all tests in the test suite
  // for Foo.
};

// Tests that the Foo::Bar() method does Abc.
TEST_F(FooTest, MethodBarDoesAbc) {
  const std::string input_filepath = "this/package/testdata/myinputfile.dat";
  const std::string output_filepath = "this/package/testdata/myoutputfile.dat";
```

```
  Foo f;
  EXPECT_EQ(f.Bar(input_filepath, output_filepath), 0);
}

// Tests that Foo does Xyz.
TEST_F(FooTest, DoesXyz) {
  // Exercises the Xyz feature of Foo.
}

}  // namespace
}  // namespace project
}  // namespace my

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```