Contents

A quick introduction to the Google C++ Testing Framework

Learn about key features for ease of use and production-level deployment

Arpan Sen
Published on May 11, 2010

4

Why use the Google C++ Testing Framework?

There are many good reasons for you to use this framework. This section describes several of them.

Some categories of tests have bad memory problems that surface only during certain runs. Google's test framework provides excellent support for handling such situations. You can repeat the same test a thousand times using the Google framework. At

Contents

Contrary to a lot of other testing frameworks, Google' amework has built-in assertions that are deployable in software where exception handling is disabled (typically for pe nce reasons). Thus, the assertions can be used safely in destructors, too.

Running the tests is simple. Just making a call to the ned `RUN_ALL_TESTS` macro does the trick, as opposed to creating or deriving a separate runner class for test execution. in sharp contrast to frameworks such as CppUnit.

Generating an Extensible Markup Language (XML) rep s easy as passing a switch: `--gtest_output="xml:<file name>"`. In frameworks such as CppUnit and CppTest, you nee te substantially more code to generate XML output.

Creating a basic test

Consider the prototype for a simple square root funct wn in Listing 1.

Listing 1. Prototype of the square root function

```
double square-root (const double);
```

For negative numbers, this routine returns `-1`. It's useful to have both positive and negative tests here, so you do both. Listing 2 shows that test case.

Listing 2. Unit test for the square root function

```
#include "gtest/gtest.h"

```

```
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}
```

Listing 2 creates a test hierarchy named SquareRootT d then adds two unit tests, PositiveNos and ZeroAndNegativeNos, to that hierarchy. TEST is a prec macro defined in gtest.h (available with the downloaded sources) that helps define this hierarchy. EXPECT_EQ and ASSER e also macros—in the former case test execution continues even if there is a failure while in the latter case test executi ts. Clearly, if the square root of 0 is anything but 0, there isn't much left to test anyway. That's why the ZeroAndNega s test uses only ASSERT_EQ while the PositiveNos test uses EXPECT_EQ to tell you how many cases there are wher quare root function fails without aborting the test.

Running the first test

Now that you've created your first basic test, it is time to run it. Listing 3 is the code for the main routine that runs the test.

Listing 3. Running the square root test

```
#include "gtest/gtest.h"

TEST(SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
```

```

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

The ::testing::InitGoogleTest method does what ne suggests—it initializes the framework and must be called before RUN_ALL_TESTS. RUN_ALL_TESTS must be callec ınce in the code because multiple calls to it conflict with some of the advanced features of the framework and, therefor ıot supported. Note that RUN_ALL_TESTS automatically detects and runs all the tests defined using the TEST macro. B ılt, the results are printed to standard output. Listing 4 shows the output.

Listing 4. Output from running the square root test

```
Running main() from user_main.cpp
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp(6862): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (9 ms)
[ RUN      ] SquareRootTest.ZeroAndNegativeNos
[       OK ] SquareRootTest.ZeroAndNegativeNos (0 ms)
[----------] 2 tests from SquareRootTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (10 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos

 1 FAILED TEST
```

Options for the Google C+ esting Framework

Contents

In Listing 3 you see that the `InitGoogleTest` function s the arguments to the test infrastructure. This section discusses some of the cool things that you can do with the argur o the testing framework.

You can dump the output into XML format by passing t_output="xml:report.xml" on the command line. You can, of course, replace `report.xml` with whatever file name y fer.

There are certain tests that fail at times and pass at m ıer times. This is typical of problems related to memory corruption. There's a higher probability of detecting tl the test is run a couple times. If you pass `--gtest_repeat=2 --gtest_break_on_failure` on the command line, the s st is repeated twice. If the test fails, the debugger is automatically invoked.

Not all tests need to be run at all times, particularly if making changes in the code that affect only specific modules. To support this, Google provides `--gtest_filter=<test` g>. The format for the test string is a series of wildcard patterns separated by colons (:). For example, `--gtest_filte` s all tests while `--gtest_filter=SquareRoot*` runs only the `SquareRootTest` tests. If you want to run only the positive unit tests from `SquareRootTest`, use `--gtest_filter=SquareRootTest.*-SquareRootTest.Zero*`. Note that `SquareRootTest.*` means all tests belonging to `SquareRootTest`, and `-SquareRootTest.Zero*` means don't run those tests whose names begin with Zero.

Listing 5 provides an example of running `SquareRootTest` with `gtest_output`, `gtest_repeat`, and `gtest_filter`.

Listing 5. Running `SquareRootTest` with `gtest_output`, `gtest_repeat`, and `gtest_filter`

```
[arpan@tintin] ./test_executable --gtest_output="xml:report.xml" --gtest_repeat=2 --
gtest_filter=SquareRootTest.*-SquareRootTest.Zero*

```

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp (6854): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (2 ms)
[----------] 1 test from SquareRootTest (2 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (20 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos
 1 FAILED TEST

Repeating all tests (iteration 2) . . .

Note: Google Test filter = SquareRootTest.*-SquareRootTest.Z*
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp (6854): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (2 ms)
[----------] 1 test from SquareRootTest (2 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (20 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos
 1 FAILED TEST
```

Comments

Let's say you break the code. Can you disable a test te rily? Yes, simply add the `DISABLE_ prefix` to the logical test name or the individual unit test name and it won't exe sting 6 demonstrates what you need to do if you want to disable the `PositiveNos` test from Listing 2.

Listing 6. Disabling a test temporarily

```
#include "gtest/gtest.h"

TEST (DISABLE_SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

OR

TEST (SquareRootTest, DISABLE_PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}
```

Note that the Google framework prints a warning at the end of the test execution if there are any disabled tests, as shown in Listing 7.

Listing 7. Google warns user of disabled tests in the framework

```
1 FAILED TEST
  YOU HAVE 1 DISABLED TEST
```

Listing 8. Google lets you run tests that are otherwise dis

```
[----------] 1 test from DISABLED_SquareRootTest
[ RUN      ] DISABLED_SquareRootTest.PositiveNos
..\user_sqrt.cpp(6854): error: Value of: square-root (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] DISABLED_SquareRootTest.PositiveNos (2 ms)
[----------] 1 test from DISABLED_SquareRootTest (2 ms total)

[  FAILED  ] 1 tests, listed below:
[  FAILED  ] SquareRootTest. PositiveNos
```

It's all about assertions

The Google test framework comes with a whole host (efined assertions. There are two kinds of assertions—those with names beginning with ASSERT_ and those beginning w ECT_. The ASSERT_* variants abort the program execution if an assertion fails while EXPECT_* variants continue with the run. In either case, when an assertion fails, it prints the file name, line number, and a message that you can customize. Some of the simpler assertions include ASSERT_TRUE (condition) and ASSERT_NE (val1, val2). The former expects the condition to always be true while the latter expects the two values to be mismatched. These assertions work on user-defined types too, but you must overload the corresponding comparison operator (==, !=, <=, and so on).

Floating point comparisons

Listing 9. Macros for floating point comparisons

```
ASSERT_FLOAT_EQ (expected, actual)
ASSERT_DOUBLE_EQ (expected, actual)
ASSERT_NEAR (expected, actual, absolute_range)

EXPECT_FLOAT_EQ (expected, actual)
EXPECT_DOUBLE_EQ (expected, actual)
EXPECT_NEAR (expected, actual, absolute_range)
```

Why do you need separate macros for floating point c sons? Wouldn't ASSERT_EQ work? The answer is that ASSERT_EQ and related macros may or may not work, and it's sma use the macros specifically meant for floating point comparisons. Typically, different central processing units (CPUs) an ting environments store floating points differently and simple comparisons between expected and actual values do k. For example, ASSERT_FLOAT_EQ (2.00001, 2.000011) passes —Google does not throw an error if the results tally up decimal places. If you want greater precision, use ASSERT_NEAR (2.00001, 2.000011, 0.0000001) and you receive t shown in Listing 10.

Listing 10. Error message from ASSERT_NEAR

```
Math.cc(68): error: The difference between 2.00001 and 2.000011 is 1e-006, which exceeds
0.0000001, where
2.00001 evaluates to 2.00001,
2.000011 evaluates to 2.00001, and
0.0000001 evaluates to 1e-007.
```

Death tests

routine or if the process exits with a proper exit code. ımple, in Listing 3, it would be good to receive an error message
when doing square-root (-22.0) and exiting the prc ⁄ith return status -1 instead of returning -1.0. Listing 11 uses
ASSERT_EXIT to verify such a scenario.

Contents

Introduction

Why use the Google C++ Testing Framework?

Listing 11. Running a death test using Google's framewor

```
#include "gtest/gtest.h"

double square-root (double num) {
    if (num < 0.0) {
        std::cerr << "Error: Negative Input\n";
        exit(-1);
    }
    // Code for 0 and +ve numbers follow

}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EXIT (square-root (-22.0), ::testing::ExitedWithCode(-1), "Error:
Negative Input");
}

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

Downloadable resources

Related topics

Comments

ASSERT_EXIT checks if the function is exiting with a proper exit code (that is, the argument to exit or _exit routines) and compares the string within quotes to whatever the function prints to standard error. Note that the error messages must go tc std::cerr and not std::cout. Listing 12 provides the prototypes for ASSERT_DEATH and ASSERT_EXIT.

Listing 12. Prototypes for death assertions

Google provides the predefined predicate `::testing:` `edWithCode(exit_code)`. The result of this predicate is true only if the program exits with the same `exit_code` mentione predicate. `ASSERT_DEATH` is simpler than `ASSERT_EXIT`; it just compares the error message in standard error with wl is the user-expected message.

Understanding test fixture

It is typical to do some custom initialization work befc cuting a unit test. For example, if you are trying to measure the time/memory footprint of a test, you need to put som pecific code in place to measure those values. This is where fixtures come in—they help you set up such custom te eeds. Listing 13 shows what a fixture class looks like.

Listing 13. A test fixture class

```
class myTestFixture1: public ::testing::test {
public:
   myTestFixture1( ) {
       // initialization code here
   }

   void SetUp( ) {
       // code here will execute just before the test ensues
   }

   void TearDown( ) {
       // code here will be called just after the test completes
       // ok to through exceptions from here if need be
   }

   ~myTestFixture1( )  {
       // cleanup any pending stuff, but no exceptions allowed
   }
```

Contents

The fixture class is derived from the `::testing::test` declared in `gtest.h`. Listing 14 is an example that uses the fixture class. Note that it uses the `TEST_F` macro instead of `T`

Listing 14. Sample use of a fixture

```
TEST_F (myTestFixture1, UnitTest1) {

.
}

TEST_F (myTestFixture1, UnitTest2) {

.
}
```

There are a few things that you need to understand w ng fixtures:

- You can do initialization or allocation of resources in either the constructor or the `SetUp` method. The choice is left to you, the user.
- You can do deallocation of resources in `TearDown` or the destructor routine. However, if you want exception handling you must do it only in the `TearDown` code because throwing an exception from the destructor results in undefined behavior.
- The Google assertion macros may throw exceptions in platforms where they are enabled in future releases. Therefore, it's a good idea to use assertion macros in the `TearDown` code for better maintenance.
- The same test fixture is *not* used across multiple tests. For every new unit test, the framework creates a new test fixture. So in Listing 14, the `SetUp` (please use proper spelling here) routine is called twice because two `myFixture1` objects are

Conclusion

This article just scratches the surface of the Google C ing Framework. Detailed documentation about the framework is available from the Google site. For advanced develope commend you read some of the other articles about open regression frameworks such as the Boost unit test fra k and CppUnit.

Downloadable resources

PDF of this content

Related topics

Google TestPrimer

Google TestAdvancedGuide

Google TestFAQ

Open source C/C++ unit testing tools, Part 1: Get to know the Boost unit test framework

What Every Computer Scientist Should Know About Floating-Point Arithmetic

Comments

Contents

□ Subscribe me to comment notifications

Select a language

English

中文

日本語

Русский

Português (Brasil)

Español

한글

Tutorials & training

Demos & sample code

Q&A forums

dW Blog

Events

Courses

Open source projects