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

Why use the Google C++ Testing Framework?

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

Some categories of tests have bad memory problems that surface only during certain runs. C
excellent support for handling such situations. You can repeat the same test a thousand time
the first sign of a failure, the debugger is automatically invoked. In addition, all of this is done
from command line: `--gtest_repeat=1000 --gtest_break_on_failure`.

Contrary to a lot of other testing frameworks, Google's test framework has built-in assertions
where exception handling is disabled (typically for performance reasons). Thus, the assertio
destructors, too.

Running the tests is simple. Just making a call to the predefined `RUN_ALL_TESTS` macro does
or deriving a separate runner class for test execution. This is in sharp contrast to frameworks

Generating an Extensible Markup Language (XML) report is as easy as passing a switch: `--gt`
In frameworks such as CppUnit and CppTest, you need to write substantially more code to g

Creating a basic test

Listing 1. Prototype of the square root function

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

For negative numbers, this routine returns -1. It's useful to have both positive and negative t
2 shows that test case.

Listing 2. Unit test for the square root function

```
1  #include "gtest/gtest.h"
2
3  TEST (SquareRootTest, PositiveNos) {
4      EXPECT_EQ (18.0, square-root (324.0));
5      EXPECT_EQ (25.4, square-root (645.16));
6      EXPECT_EQ (50.3321, square-root (2533.310224));
7  }
8
9  TEST (SquareRootTest, ZeroAndNegativeNos) {
10     ASSERT_EQ (0.0, square-root (0.0));
11     ASSERT_EQ (-1, square-root (-22.0));
12 }
```

Listing 2 creates a test hierarchy named SquareRootTest and then adds two unit tests, Posi
ZeroAndNegativeNos, to that hierarchy. TEST is a predefined macro defined in gtest.h (availa
that helps define this hierarchy. EXPECT_EQ and ASSERT_EQ are also macros—in the former ca
if there is a failure while in the latter case test execution aborts. Clearly, if the square root of
much left to test anyway. That's why the ZeroAndNegativeNos test uses only ASSERT_EQ whi
EXPECT_EQ to tell you how many cases there are where the square root function fails without

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 n

Listing 3. Running the square root test

```
1  #include "gtest/gtest.h"
2
3  TEST(SquareRootTest, PositiveNos) {
4      EXPECT_EQ (18.0, square-root (324.0));
5      EXPECT_EQ (25.4, square-root (645.16));
6      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 the name suggests—it initializes the fra
before RUN_ALL_TESTS. RUN_ALL_TESTS must be called only once in the code because multip
the advanced features of the framework and, therefore, are not supported. Note that RUN_AL
and runs all the tests defined using the TEST macro. By default, the results are printed to sta
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++ Testing Framewo

In Listing 3 you see that the InitGoogleTest function accepts the arguments to the test infr
some of the cool things that you can do with the arguments to the testing framework.

You can dump the output into XML format by passing --gtest_output="xml:report.xml" o
course, replace report.xml with whatever file name you prefer.

There are certain tests that fail at times and pass at most other times. This is typical of probl
corruption. There's a higher probability of detecting the fail if the test is run a couple times. I

Not all tests need to be run at all times, particularly if you are making changes in the code th
support this, Google provides `--gtest_filter=<test string>`. The format for the test strin
separated by colons (:). For example, `--gtest_filter=*` runs all tests while `--gtest_filte`
`SquareRootTest` tests. If you want to run only the positive unit tests from `SquareRootTest`, u
`gtest_filter=SquareRootTest.*-SquareRootTest.Zero*`. Note that `SquareRootTest.*` m
`SquareRootTest`, and `-SquareRootTest.Zero*` means don't run those tests whose names b

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

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

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

Repeating all tests (iteration 1) . . .

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

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:
```

```
41 [  FAILED  ] SquareRootTest.PositiveNos
42  1 FAILED TEST
```


Temporarily disabling tests

Let's say you break the code. Can you disable a test temporarily? Yes, simply add the `DISABL` name or the individual unit test name and it won't execute. Listing 6 demonstrates what you the `PositiveNos` test from Listing 2.

Listing 6. Disabling a test temporarily

```
1  #include "gtest/gtest.h"
2
3  TEST (DISABLE_SquareRootTest, PositiveNos) {
4      EXPECT_EQ (18.0, square-root (324.0));
5      EXPECT_EQ (25.4, square-root (645.16));
6      EXPECT_EQ (50.3321, square-root (2533.310224));
7  }
8
9  OR
10
11 TEST (SquareRootTest, DISABLE_PositiveNos) {
12     EXPECT_EQ (18.0, square-root (324.0));
13     EXPECT_EQ (25.4, square-root (645.16));
14     EXPECT_EQ (50.3321, square-root (2533.310224));
15 }
```

Note that the Google framework prints a warning at the end of the test execution if there are Listing 7.

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

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

If you want to continue running the disabled tests, pass the `-gtest_also_run_disabled_te` Listing 8 shows the output when the `DISABLE_PositiveNos` test is run.

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

```
1  [----------] 1 test from DISABLED_SquareRootTest
2  [ RUN      ] DISABLED_SquareRootTest.PositiveNos
3  ..\user_sqrt.cpp(6854): error: Value of: square-root (2533.310224)
4    Actual: 50.332
5  Expected: 50.3321
6  [  FAILED  ] DISABLED_SquareRootTest.PositiveNos (2 ms)
7  [----------] 1 test from DISABLED_SquareRootTest (2 ms total)
8
```

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


It's all about assertions

Contents

The Google test framework comes with a whole host of predefined assertions. There are two names beginning with `ASSERT_` and those beginning with `EXPECT_`. The `ASSERT_*` variants ab assertion fails while `EXPECT_*` variants continue with the run. In either case, when an asserti line number, and a message that you can customize. Some of the simpler assertions include `ASSERT_NE (val1, val2)`. The former expects the condition to always be true while the latt mismatched. These assertions work on user-defined types too, but you must overload the co (==, !=, <=, and so on).

Floating point comparisons

Google provides the macros shown in Listing 9 for 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 comparisons? Wouldn't `ASSERT_EQ` work and related macros may or may not work, and it's smarter to use the macros specifically mea Typically, different central processing units (CPUs) and operating environments store floatin comparisons between expected and actual values don't work. For example, `ASSERT_FLOAT_E` —Google does not throw an error if the results tally up to four decimal places. If you want gre `(2.00001, 2.000011, 0.0000001)` and you receive the error 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, wh
0.0000001, where
2.00001 evaluates to 2.00001,
2.000011 evaluates to 2.00001, and
0.0000001 evaluates to 1e-007.
```

Death tests

calls the *death assertions*. You use this type of assertion to check if a proper error message i routine or if the process exits with a proper exit code. For example, in Listing 3, it would be g when doing `square-root (-22.0)` and exiting the program with return status `-1` instead of r `ASSERT_EXIT` to verify such a scenario.

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

```
#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();
}
```

`ASSERT_EXIT` checks if the function is exiting with a proper exit code (that is, the argument to compares the string within quotes to whatever the function prints to standard error. Note tha `std::cerr` and not `std::cout`. Listing 12 provides the prototypes for `ASSERT_DEATH` and `ASS`

Listing 12. Prototypes for death assertions

```
ASSERT_DEATH(statement, expected_message)
ASSERT_EXIT(statement, predicate, expected_message)
```

Google provides the predefined predicate `::testing::ExitedWithCode(exit_code)`. The re the program exits with the same `exit_code` mentioned in the predicate. `ASSERT_DEATH` is sin compares the error message in standard error with whatever is the user-expected message.

Understanding test fixtures

It is typical to do some custom initialization work before executing a unit test. For example, i

Listing 13. A test fixture class

Contents

```
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
   }

   // put in any custom data members that you need
};
```

Understanding test fixtures

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

Conclusion

Downloadable resources

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 when using fixtures:

- You can do initialization or allocation of resources in either the constructor or the `SetUp` r the user.
- You can do deallocation of resources in `TearDown` or the destructor routine. However, if y must do it only in the `TearDown` code because throwing an exception from the destructor

- The Google assertion macros may throw exceptions in platforms where they are enabled a good idea to use assertion macros in the `TearDown` code for better maintenance.

So in Listing 14, the `SetUp` (please use proper spelling here) routine is called twice becau created.

Conclusion

This article just scratches the surface of the Google C++ Testing Framework. Detailed docum available from the Google site. For advanced developers, I recommend you read some of the regression frameworks such as the Boost unit test framework and CppUnit.

Downloadable resources

PDF of this content

Related topics

Understanding test fixtures

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

Sign in or register to add and subscribe to comments.

Subscribe me to comment notifications