GTest - Quick How To

Dec. 2021 - GoogleTests v1.11

1 Adding a new class test to CMake

Make sure there is an executable source with the following instructions:

```
// file: launch_tests.cpp
#include "gtest/gtest.h"      // COMPULSORY

using namespace std;
using namespace testing;

int main(int argc, char **argv)
{
  testing::InitGoogleTest(&argc, argv); // COMPULSORY
  return RUN_ALL_TESTS();    // COMPULSORY, called only ONCE
}
```

Note: The arguments are compulsory to pass options from the CMakefile to GTest.

In the `CMakeLists.txt`, you can add the following lines:

```
add_executable(runUnitTests launch_tests.cpp myClass_UT.cpp)
```

where the file `myClass_UT.cpp` will contain your new tests.

Declare your dependencies and then add those to the GoogleTests library by using:

```
target_link_libraries(runUnitTests PUBLIC ${CMAKE_PROJECT_NAME}
                                          gtest
                                          gtest_main)
```

Add `pthread` to the list if your code run with multi-threading.

In order to create a special command that will run only your new executable `runUnitTests`, add also in the `CMakeLists.txt`:

```
   add_test(NAME runUnitTests
            COMMAND runUnitTests_cmd)
```

In your Terminal, you can now call:

```
> mkdir build
> cd ./build
> cmake .. && make runUnitTests
> cd ./tests
> ./runUnitTests --gtest_output="xml:report.xml"
```

The option `--gtest_output="xml:report.xml"` provides a .xml report compatible with Gitlab CI that describes the passed and failing tests. It is mandatory to validate the successful status of the pipeline.

2 Writing the new class test

In your file `CMakeLists.txt`, you can now write create your test suite.

2.1 Simple TEST

```
// file: myClass_UT.cpp
#include "gtest/gtest.h" // COMPULSORY for each new file
#include "myClass.cpp"

using namespace testing;

class myClassTest : public ::testing::Test
{
protected:
  myClassTest() {}      // COMPULSORY
  ~myClassTest() {}     // COMPULSORY
};

TEST(myClassTest, worksWithFunc1)
{
  // Declare your variables here
  // Make use of myClass::func1
  // Compare the resulting value(s) from your expectation(s)
  ASSERT_EQ(resulting_value, expected_value)
            << "Msg to display in case of failure for func1";
}

TEST(myClassTest, worksWithFunc2)
{
  // Declare your variables here
  // Make use of myClass::func2
  // If you already have a boolean:
  ASSERT_TRUE(myBool)
            << "Msg to display in case of failure for func2";
  ASSERT_FALSE(myBool)
            << "Msg to display in case of failure for func2";
}
```

Once this skeleton is in place, you can add as much tests as you need for the member functions of myClass.

If some of your tests require the exact same data configuration for each test, you can use test fixtures.

2.2 Test Fixtures: TEST_F

For each test, the data configuration declared in the `SetUp` will be reset to those values, no matter what has been modified in the previous tests: there is no need for global variables. Those SetUp values can also be inherited.

```
// file: myClass_UT.cpp
#include "gtest/gtest.h" // COMPULSORY for each new file
#include "myClass.cpp"

using namespace testing;

class myClassTest : public ::testing::Test
{
protected:
  myClassTest() {}      // COMPULSORY

  // Declare your variable(s) here
  myClass obj1;
  ...

  // Assign values to your variables for the whole test set
  virtual void SetUp() override
  {
     obj1 = value_x;
     ...
  }
  // Free your variables if need be
  virtual void TearDown() override {}

  ~myClassTest() {}     // COMPULSORY
};
```

```
// TEST_F = Test Fixture, which is allowed to used values in SetUp
TEST_F(myClassTest, worksWithFunc1)
{
  // Declare your other variables here
  // Make use of myClass::func1 with obj1 e.g.
  // Compare the resulting value(s) from your expectation(s)
  EXPECT_EQ(resulting_value, expected_value)
            << "Msg to display in case of failure for func1";
}
```

Warnings: GoogleTest prevents the mix of TEST and TEST_F!

2.2.1 Test Fixtures: The constructor vs the SetUp function

Each time a test is called, the constructor and the SetUp will be called - and if variables had their values set in one of those functions, they will therefore be reset to those values before the next test.

Then why using the SetUp function? It is safer if an exception is thrown by one of the functions called inside of it. The constructor of the test class will not be able to dealt properly with it, and this may lead to crash.

3 Options

For more examples, please see the GitHub for GoogleTest, in particular their "samples" folder: `https://github.com/google/googletest/tree/master/googletest/samples`.

There are more advanced use of GoogleTest, described here: `https://google.github.io/googletest/advanced.html`.

For more variations on `ASSERT` and `EXPECT` commands, please see the Google Tests documentation at: `https://google.github.io/googletest/reference/assertions.html`.