Parameterized testing with GTest

Sandor Dargo

MUREX

CODE RECKONS

Science to the CORE

Who Am I?

Sándor DARGÓ

- Principal Engineer in Amadeus
- Enthusiastic blogger https://www.sandordargo.com
- (A former) Passionate traveller
- Curious home baker
- Happy father of two

Agenda

How to make our tests less repetitive without parameterized tests?

Parameterized tests, what are they?

Write parameterized tests by scratch

Write parameterized tests based on an existing fixture

How to pass multiple parameters to the same test case?

Bonus: type-parameterized tests!

Why this talk?

Handle your test code as production code and keep it DRY

The idea of parameterized testing is great

Documentation is scarce

Our training ground for today is the leap year kata

Check the code at:

https://github.com/sandordargo/parameterizedTestExamplesCpp

Read my article on parameterized tests at:

https://www.sandordargo.com/blog/2019/04/24/parameterized-testing-with-gtest

GTest user guide is at:

https://google.github.io/googletest/

Example of some repetitive test

- Verbose
- Repeated code
+ Descriptive

+ Helpful error messages

```
TEST(LeapYearTests, 1IsOdd_IsNotLeapYear) {
  LeapYearCalendar leapYearCalendar;
  ASSERT_FALSE(leapYearCalendar.isLeap(1));
}

TEST(LeapYearTests, 711IsOdd_IsNotLeapYear) {
  LeapYearCalendar leapYearCalendar;
  ASSERT_FALSE(leapYearCalendar.isLeap(711));
}

TEST(LeapYearTests, 1989IsOdd_IsNotLeapYear) {
  LeapYearCalendar leapYearCalendar;
  ASSERT_FALSE(leapYearCalendar.isLeap(1989));
}

TEST(LeapYearTests, 2013IsOdd_IsNotLeapYear) {
  LeapYearCalendar leapYearCalendar;
  ASSERT_FALSE(leapYearCalendar.isLeap(2013));
}
```

When a test case fails...

You get detailed error messages

```
[----------] 4 tests from LeapYearTests
[ RUN      ] LeapYearTests.1IsOdd_IsNotLeapYear
[       OK ] LeapYearTests.1IsOdd_IsNotLeapYear (0 ms)
[ RUN      ] LeapYearTests.711IsOdd_IsNotLeapYear
[       OK ] LeapYearTests.711IsOdd_IsNotLeapYear (0 ms)
[ RUN      ] LeapYearTests.1989IsOdd_IsNotLeapYear
/home/sdargo/personal/dev/LeapYear/tests/
LeapYearStandaloneTests.cpp:17: Failure
Value of: leapYear.isLeap(1989)
  Actual: true
Expected: false
[  FAILED  ] LeapYearTests.1989IsOdd_IsNotLeapYear (0 ms)
[ RUN      ] LeapYearTests.2013IsOdd_IsNotLeapYear
[       OK ] LeapYearTests.2013IsOdd_IsNotLeapYear (0 ms)
[----------] 4 tests from LeapYearTests (0 ms total)
```

Limit repetition with a fixture

- \- Repeated logic
- \+ Less verbose
- \+ Still descriptive

- \+ The same helpful error messages

```
class LeapYearFixtureTests :
  public ::testing::Test {
 protected:
    LeapYearCalendar leapYearCalendar;
};

TEST_F(LeapYearFixtureTests,
       1IsOdd_IsNotLeapYear) {
    ASSERT_FALSE(leapYearCalendar.isLeap(1));
}

TEST_F(LeapYearFixtureTests,
       711IsOdd_IsNotLeapYear) {
    ASSERT_FALSE(leapYearCalendar.isLeap(711));
}
```

The good old for loop

- Not very descriptive
+ No repetition

- Meaningless error messages

```
TEST(LeapYearIterationTest,
     OddYearsAreNotLeapYears) {
  LeapYearCalendar leapYearCalendar;
  std::vector oddYears = {1, 711, 1989, 2013};
  for (auto oddYear:  oddYears) {
    ASSERT_FALSE(
      leapYearCalendar.isLeap(oddYear)
    );
  }
}
```

When a test case fails...

You have no idea which input invokes the error

```
[----------] 1 test from LeapYearIterationTest
[ RUN      ] LeapYearIterationTest.OddYearsAreNotLeapYears
/home/sdargo/personal/dev/LeapYear/tests/
LeapYearIterationTest.cpp:9: Failure
Value of: leapYear.isLeap(oddYear)
  Actual: true
Expected: false
[  FAILED  ] LeapYearIterationTest.OddYearsAreNotLeapYears (0 ms)
[----------] 1 test from LeapYearIterationTest (0 ms total)
```

What does parameterized testing bring?

Less repetitive test code

Meaningful error messages

Easy execution of the same logic with several inputs

No silver bullet, but a good addition to your toolbox

Parameterized tests without a fixture

Inherit from `TestWithParam<T>`

You can move some setup to the class initialization

```
class LeapYearParameterizedTestFixture :
        public ::testing::TestWithParam<int> {
 protected:
    LeapYearCalendar leapYearCalendar;
};
```

Parameterized tests without a fixture

Use the `TEST_P` macro to describe your parameterized test

Use `GetParam()` to read the parameter value of the current iteration

```
class LeapYearParameterizedTestFixture :
        public ::testing::TestWithParam<int> {
 protected:
    LeapYearCalendar leapYearCalendar;
};

TEST_P(LeapYearParameterizedTestFixture,
       OddYearsAreNotLeapYears) {
    int year = GetParam();
    ASSERT_FALSE(leapYearCalendar.isLeap(year));
}
```

Parameterized tests without a fixture

`INSTANTIATE_TEST_SUITE_P` to pass in the parameters

Before v1.10 it used to be *`INSTANTIATE_TEST_CASE_P`*

```
class LeapYearParameterizedTestFixture :
        public ::testing::TestWithParam<int> {
 protected:
    LeapYearCalendar leapYearCalendar;
};

TEST_P(LeapYearParameterizedTestFixture,
       OddYearsAreNotLeapYears) {
    int year = GetParam();
    ASSERT_FALSE(leapYearCalendar.isLeap(year));
}

INSTANTIATE_TEST_SUITE_P(
    LeapYearTests,
    LeapYearParameterizedTestFixture,
    ::testing::Values(1, 711, 1989, 2013)
);
```

When a test case fails...

You get all the relevant details

```
[----------] 2 tests from LeapYearTests/LeapYearParamTests
[ RUN      ] LeapYearTests/LeapYearParamTests.OddYearsAreNotLeapYears/0
/home/sdargo/personal/dev/LeapYear/tests/
      LeapYearParameterizedTestFixture.cpp:12: Failure
Value of: leapYear.isLeap(year)
  Actual: true
Expected: false
[  FAILED  ] LeapYearTests/LeapYearParamTests.OddYearsAreNotLeapYears/0,
             where GetParam() = 1989 (0 ms)
[ RUN      ] LeapYearTests/LeapYearParamTests.OddYearsAreNotLeapYears/1
[       OK ] LeapYearTests/LeapYearParamTests.OddYearsAreNotLeapYears/1
(0 ms)
[----------] 2 tests from LeapYearTests/LeapYearParamTests (0 ms total)
```

Write parameterized tests based on an existing fixture

There might be already a setup that you want reuse

You can keep your existing fixture

```
class LeapYearTestFixture : public ::testing::Test {
 protected:
  LeapYearCalendar leapYearCalendar;
};

TEST_F(LeapYearTestFixture,
       1996_DivisibleBy4_LeapYear) {
  ASSERT_TRUE(leapYearCalendar.isLeap(1996));
}

TEST_F(LeapYearTestFixture,
       1600_IsDivisibleBy400_LeapYear) {
  ASSERT_TRUE(leapYearCalendar.isLeap(1600));
}
```

Write parameterized tests based on an existing fixture

Additional class inheriting from the fixture

Also inherit from

`::testing::WithParamInterface<T>`

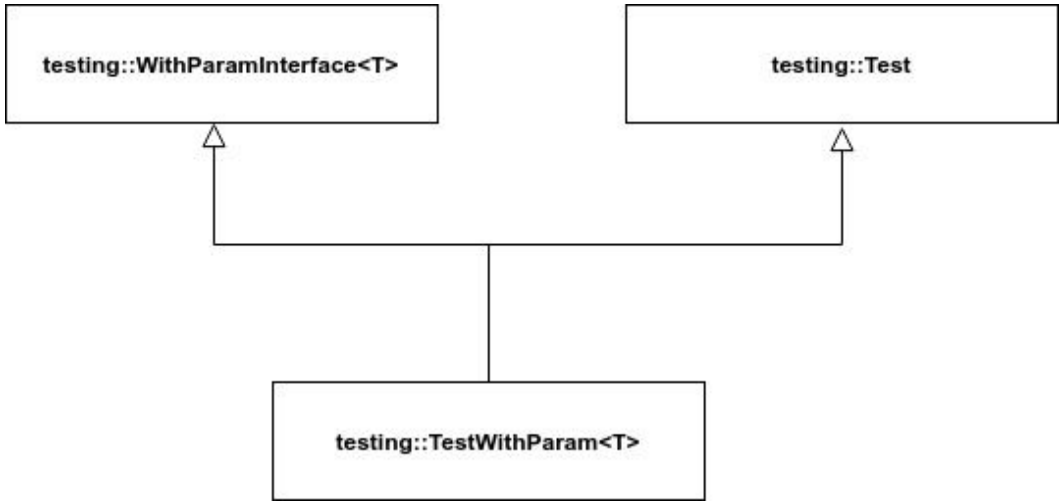
```
class LeapYearTestFixture :
        public ::testing::Test {
 protected:
  LeapYearCalendar leapYearCalendar;
};

class LeapYearParametrizedTestsBasedOnFixture :
  public LeapYearTestFixtureToBeParameterized,
  public ::testing::WithParamInterface<int> {
};
```

Why the base class changed?

To avoid compilation failure:

```
error:
 'testing::Test' is an
 ambiguous base of
 'LeapYearParamTests
 BasedOnFixture_
 ChecksIfLeapYear_Test'
```



`TestWithParam<T>` already inherits from `Test`

What if your test needs multiple parameters?

Your tests might have more than one changing input

`TestWithParam<T>` takes only one template argument!

You can pass in any type, but the usual choice is a `std::tuple`

`TestWithParam<T>` with multiple parameters

Pass the type to `TestWithParam<T>`

```
class LeapYearMultiParamTests :
  public ::testing::TestWithParam<
    std::tuple<int, bool>> {
 protected:
    LeapYearCalendar leapYearCalendar;
};
```

`TestWithParam<T>` with multiple parameters

Use `GetParam()` combined with `std::get<N>` to read each parameter

Or use *structured bindings* since C++17

```
class LeapYearMultiParamTests :
  public ::testing::TestWithParam<
    std::tuple<int, bool>> {
 protected:
    LeapYearCalendar leapYearCalendar;
};

TEST_P(LeapYearMultiParamTests,
       ChecksIfLeapYear) {
    bool expected = std::get<1>(GetParam());
    int year = std::get<0>(GetParam());
    // auto [year, expected] = GetParam(); // C++17!
    ASSERT_EQ(expected,
              leapYearCalendar.isLeap(year));
}
```

`TestWithParam<T>` with multiple parameters

Create and pass tuples as `Values`

```
INSTANTIATE_TEST_CASE_P(
    LeapYearTests,
    LeapYearMultiParamTests,
    ::testing::Values(
        std::make_tuple(7, false),
        std::make_tuple(2001, false),
        std::make_tuple(1996, true),
        std::make_tuple(1700, false),
        std::make_tuple(1600, true)));
```

When a test case fails...

You get all the inputs

But the test name might be less meaningful

```
[----------] 2 tests from LeapYearTests/LeapYearMultiParamTests
[ RUN      ] LeapYearTests/LeapYearMultiParamTests.ChecksIfLeapYear/0
[       OK ] LeapYearTests/LeapYearMultiParamTests.ChecksIfLeapYear/0 (0 ms)
[ RUN      ] LeapYearTests/LeapYearMultiParamTests.ChecksIfLeapYear/1
/home/sdargo/personal/dev/LeapYear/tests/
LeapYearMultiParamTests.cpp:16: Failure
Expected equality of these values:
  Expected
    Which is: false
  leapYear.isLeap(year)
    Which is: true
[  FAILED  ] LeapYearTests/LeapYearMultiParamTests.ChecksIfLeapYear/1,
             where GetParam() = (1989, false) (0 ms)
[----------] 2 tests from LeapYearTests/LeapYearMultiParamTests (0 ms total)
```

What if you want to test different types?

You can use the combination of a variant and a visitor!

```
template <typename T, typename U>
auto add(T a, U b) {
    return a + b;
}
```

Typish parameterized tests

Use `std::variant` as a parameter!

```
class AddTypishParamTests :public
::testing::TestWithParam<std::variant<int, double>>
{};
```

Typish parameterized tests

Use a visitor to get the right type without knowing it!

```
class AddTypishParamTests :public
::testing::TestWithParam<std::variant<int, double>>
{};

TEST_P(AddTypishParamTests, doesAddNumbers) {
    std::visit([this](auto&& arg) {
        ASSERT_EQ(10, add(arg, arg));
    }, GetParam());
}
```

Typish parameterized tests

Use freely the different types as `Values`

```
class AddTypishParamTests :public
::testing::TestWithParam<std::variant<int, double>>
{};

TEST_P(AddTypishParamTests, doesAddNumbers) {
    std::visit([this](auto&& arg) {
        ASSERT_EQ(10, add(arg, arg));
    }, GetParam());
}

INSTANTIATE_TEST_SUITE_P(
        AddTests, AddTypishParamTests,
        ::testing::Values(
                5, 5.0
        ));
```

But you also have built-in support for typed tests

Some repetition to declare the typed parameterized suite

```
template<typename T>
class AddTypedParamTestsFixture : public ::testing::Test {};

TYPED_TEST_SUITE_P(AddTypedParamTestsFixture);
```

But you also have built-in support for typed tests

More macros to complete the registration

Values don't depend on the type

```
template<typename T>
class AddTypedParamTestsFixture : public ::testing::Test {};

TYPED_TEST_SUITE_P(AddTypedParamTestsFixture);

TYPED_TEST_P(AddTypedParamTestsFixture, doesAdd) {
  auto result = add<TypeParam>(5, 6);
  ASSERT_EQ(11, result);
}

REGISTER_TYPED_TEST_SUITE_P(AddTypedParamTestsFixture,
                            doesAdd);
```

But you also have built-in support for typed tests

List finally the types

```
template<typename T>
class AddTypedParamTestsFixture : public ::testing::Test {};

TYPED_TEST_SUITE_P(AddTypedParamTestsFixture);

TYPED_TEST_P(AddTypedParamTestsFixture, doesAdd) {
  auto result = add<TypeParam>(5, 6);
  ASSERT_EQ(11, result);
}

REGISTER_TYPED_TEST_SUITE_P(AddTypedParamTestsFixture,
                            doesAdd);

using Types = testing::Types<int, long long, std::size_t>;
INSTANTIATE_TYPED_TEST_SUITE_P(TestPrefix,
                               AddTypedParamTestsFixture,
                               Types);
```

What other libs are out there?

Catch2 with generators:
https://github.com/catchorg/Catch2/blob/devel/docs/generators.md

Doctest:
https://github.com/onqtam/doctest/blob/master/doc/markdown/parameterized-tests.md

Boost Test:
https://www.boost.org/doc/libs/1_64_0/libs/test/doc/html/boost_test/tests_organization/test_cases/param_test.html

Conclusion

Use parameterized tests to decrease code duplication

Use it to test the same logic extensively with several inputs

Don’t overuse, multiple parameters can decrease readability

Parameterized testing with GTest

Sandor Dargo

MUREX

CODE RECKONS

Science to the CORE