Asserting Using Catch2

Dror Helper

@dhelper blog.drorhelper.com

Overview

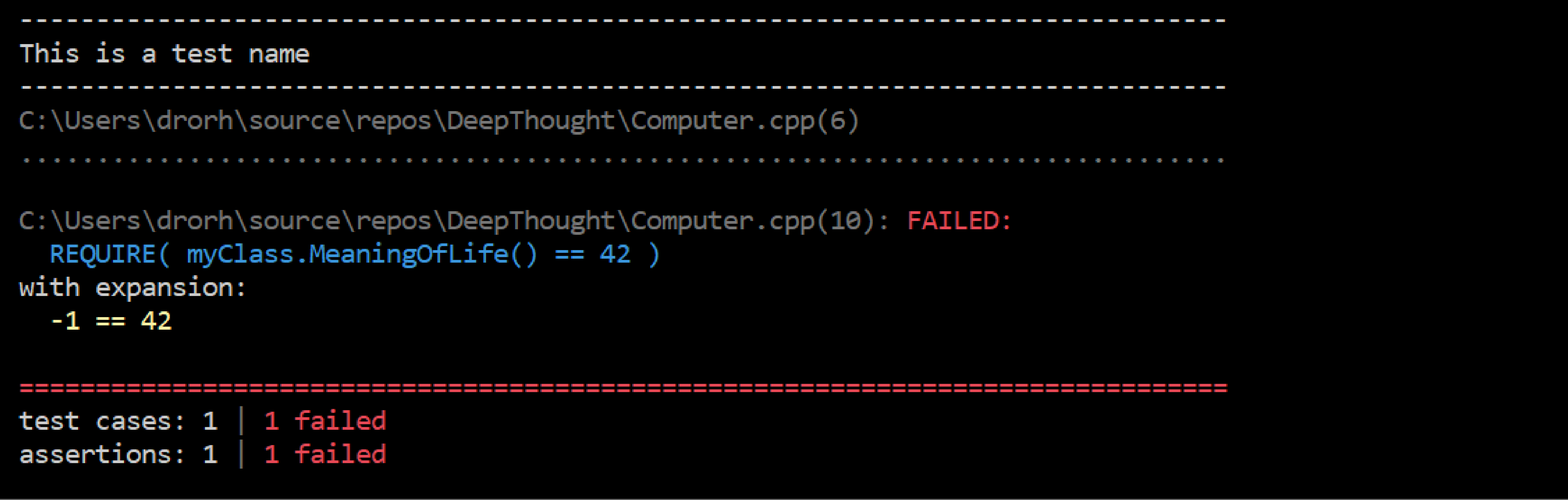
Using REQUIRE

Multiple Asserts in one test

Checking for exceptions

Adding more information to failures

Converting types into strings



```
-------------------------------------------------------------------------------
This is a test name
-------------------------------------------------------------------------------
C:\Users\drorh\source\repos\DeepThought\Computer.cpp(6)
...............................................................................

C:\Users\drorh\source\repos\DeepThought\Computer.cpp(10): FAILED:
  REQUIRE( myClass.MeaningOfLife() == 42 )
with expansion:
  -1 == 42

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

REQUIRE

Single macro for all/most assertions needs

Write the assertion in plain code

Excellent failure messages

Why You Should Care About Failure Messages?

Understand Why
The Test Failed

Reduce
Debugging Time

It’s the purpose
of the test

What’s Wrong With This Test?

```
TEST_CASE("Encode uppercase letter --> return digit")
{
    StringToDigitsEncoder encoder;

     Digits expected({ 2 });

    REQUIRE(encoder.Encode("A") == expected);
    REQUIRE(encoder.Encode("B") == expected);
    REQUIRE(encoder.Encode("C") == expected);
}
```

The problem with Multiple Assertions

Lose information

Testing more than one aspect

Create complicated tests

Multiple Assertions for a Single Result

```
TEST_CASE(“Tree has other word that begins with same letter”) {
    WordsTree tree;
    tree.AddWord("ab", { 2, 2 });
    tree.AddWord("ad", { 2, 3 });

    auto result = tree.GetWords(Digits{ 2, 3 });

    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == "ad");
}
```

When to use Multiple Assertions?

Multiple checks for single “concept”

Checking related logic

Always be pragmatic

REQUIRE and CHECK

```
REQUIRE (2 + 2 == 5);  // Abort test --> Test fail
CHECK (2 + 2 == 5);    // Continue test --> Test fail
```

~~`REQUIRE(!MethodReturnsFalse());`~~

```
REQUIRE_FALSE(MethodReturnsFalse);
CHECK_FALSE(MethodReturnsFalse);
```

Handling Multiple Assertions in One Test

Split to multiple tests

Use CHECK

Override operator ==

Compare Collections

Use Multiple asserts

Demo

Fixing existing tests

- REQUIRE vs. CHECK
- Splitting tests
- Overloading *operator==*
- Comparing collections

Asserting for Exceptions

```
REQUIRE_THROWS( expression )
CHECK_THROWS( expression )

REQUIRE_THROWS_AS( expression, type )
CHECK_THROWS_AS( expression type )

REQUIRE_NOTHROW( expression )
CHECK_NOTHROW( expression )
```

Demo

Testing for exceptions

Using Matchers

```
REQUIRE_THAT( result, matcher expression )
CHECK_THAT( result, matcher expression )

REQUIRE_THAT(numbers, VectorContains(3))
CHECK_THAT(str, StartsWith("Hello") || !EndsWith("World"))
```

String Matchers

```
REQUIRE_THAT(str, Contains("abcd"))
REQUIRE_THAT(str, StartsWith("abcd"))
REQUIRE_THAT(str, EndsWith("abcd"))
REQUIRE_THAT(str, Equals("abcd"))


REQUIRE_THAT(str, Matches("abc.*"))


REQUIRE_THAT(str, Contains("abcd", Catch::CaseSensitive::No))
```

Vector Matchers

```
REQUIRE_THAT(vec1, Contains(vec2))

REQUIRE_THAT(vec, VectorContains(1))

REQUIRE_THAT(vec1, Equals(vec2))

REQUIRE_THAT(vec1, UnorderedEquals(vec2))

REQUIRE_THAT(vec1, Approx(vec2))
```

Floating Point Matchers

```
REQUIRE_THAT(value, WithinAbs(11.0, 0.5));
REQUIRE_THAT(value, WithinULP(11.0, 2.0));
REQUIRE_THAT(value, WithinRel(11.0, 0.5));
```

Exception Matchers

```
REQUIRE_THROWS_WITH(MyFunc(), "Something bad happened")
CHECK_THROWS_WITH(MyFunc(), Contains("Something bad"))

REQUIRE_THROWS_MATCHES(MyFunc(), SomeException, matcher)
CHECK_THROWS_MATCHES(MyFunc(), SomeException, matcher)
```

Generic Matchers

```
REQUIRE_THAT(val,
            Predicate<int>(
               [](int i) -> { return i % 2 == 0; },
               "Number must be even"));
```

Custom Matchers

```
class IntMatcher : public Catch::MatcherBase<int> {
public:
    bool match( int const& i ) const override {
        // Performs the test for this matcher
    }

    virtual std::string describe() const override {
        // Produces a string describing what this matcher does
    }
};
```

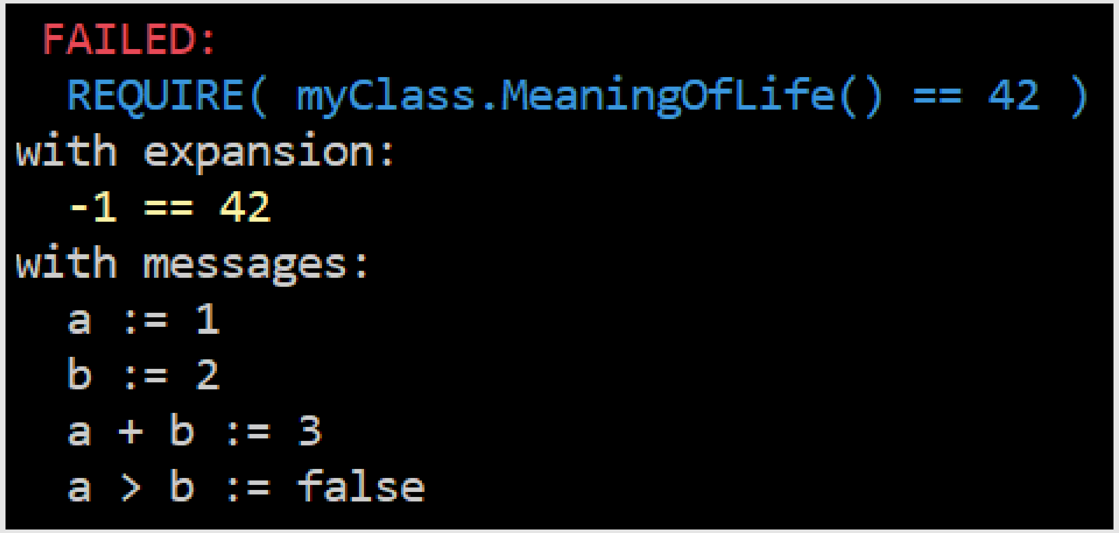
Adding More Information to Test Run

INFO	WARN	FAIL
UNSCOPED_INFO	CAPTURE	FAIL_CHECK

Logging Macros

```
INFO(“Passed first step”);
INFO(“Customer name is: ” << customer.get_name());


CAPTURE(someValue); // someValue := 123
CAPTURE(a, b, a + b, a > b);
```

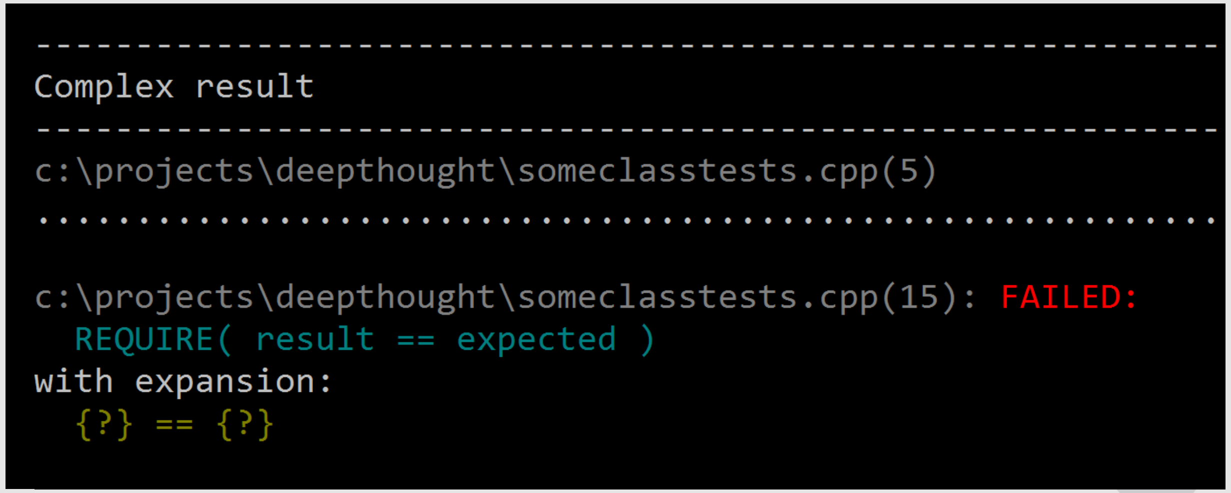


```
 FAILED:
  REQUIRE( myClass.MeaningOfLife() == 42 )
with expansion:
  -1 == 42
with messages:
  a := 1
  b := 2
  a + b := 3
  a > b := false
```

Simple information from complex types

```cpp
class SomeClass
{
public:
    int my_int_;
    double my_double_;
};
```

```cpp
REQUIRE(result == expected)
```



```
-------------------------------------------------------------------------------
Complex result
-------------------------------------------------------------------------------
c:\projects\deepthought\someclasstests.cpp(5)
...............................................................................

c:\projects\deepthought\someclasstests.cpp(15): FAILED:
  REQUIRE( result == expected )
with expansion:
  {?} == {?}
```

String Conversions

operator<<

Catch::StringMaker specialisation

CATCH_REGISTER_ENUM

CATCH_TRANSLATE_EXCEPTION

Operator << Overloading for std::ostream

```
ostream& operator<< (ostream& os, MyType const& value )
{
    os << convert ( value );
    return os;
}
```

Catch::StringMaker Specialisation

```
namespace Catch {
    template<> struct StringMaker<T> {
        static std::string convert( T const& value ) {
            return convert ( value );
        }
    };
}
```

Convert Enums to Strings

```
CATCH_REGISTER_ENUM(MyEnum, MyEnum::One, MyEnum::Two,...)
```

Custom Exception Text

```
CATCH_TRANSLATE_EXCEPTION( MyType& ex )
{
    return ex.message();
}
```

Summary

REQUIRE and CHECK

Multiple asserts in one test

Why we care about failure messages

Logging test information

Customizing the way objects are shown