Tutorial

Contents

Getting Catch2

Ideally you should be using Catch2 through its CMake integration.
Catch2 also provides pkg-config files and single TU distribution, but this documentation will assume you are using CMake. If you are using single-TU distribution instead, remember to replace the included header with `catch_amalgamated.hpp`.

Writing tests

Let's start with a really simple example (code). Say you have written a function to calculate factorials and now you want to test it (let's leave aside TDD for now).

```
unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}
```

```
#include <catch2/catch_test_macros.hpp>
unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}
TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

This will compile to a complete executable which responds to command line arguments. If you just run it with no arguments it will execute all test cases (in this case there is just one), report any failures, report a summary of how many tests passed and failed and return the number of failed tests (useful for if you just want a yes/ no answer to: "did it work").

Anyway, as the tests above as written will pass, but there is a bug.
The problem is that `Factorial(0)` should return 1 (due to its definition).
Let's add that as an assertion to the test case:

```
TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(0) == 1 );
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

After another compile & run cycle, we will see a test failure. The output will look something like:

```
Example.cpp:9: FAILED:
  REQUIRE( Factorial(0) == 1 )
with expansion:
  0 == 1
```

Note that the output contains both the original expression, `REQUIRE( Factorial(0) == 1 )` and the actual value returned by the call to the `Factorial` function: `0`.

We can fix this bug by slightly modifying the `Factorial` function to:

```
unsigned int Factorial( unsigned int number ) {
  return number > 1 ? Factorial(number-1)*number : 1;
}
```

What did we do here?

Although this was a simple test it's been enough to demonstrate a few things about how Catch2 is used. Let's take a moment to consider those before we move on.

- We introduce test cases with the `TEST_CASE` macro. This macro takes one or two string arguments - a free form test name and, optionally, one or more tags (for more see Test cases and Sections).
- The test automatically self-registers with the test runner, and user does not have do anything more to ensure that it is picked up by the test framework. *Note that you can run specific test, or set of tests, through the command line.*

- The individual test assertions are written using the `REQUIRE` macro. It accepts a boolean expression, and uses expression templates to internally decompose it, so that it can be individually stringified on test failure.

On the last point, note that there are more testing macros available, because not all useful checks can be expressed as a simple boolean expression. As an example, checking that an expression throws an exception is done with the `REQUIRE_THROWS` macro. More on that later.

Test cases and sections

Like most test frameworks, Catch2 supports a class-based fixture mechanism, where individual tests are methods on class and setup/teardown can be done in constructor/destructor of the type.

hy -> a
git -> b
ass1_h -> c.

However, their use in Catch2 is rare, because idiomatic Catch2 tests instead use *sections* to share setup and teardown code between test code. This is best explained through an example (code):

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {
    std::vector<int> v( 5 );
    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );
    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );
        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "resizing smaller changes size but not capacity" ) {
        v.resize( 0 );
        REQUIRE( v.size() == 0 );
        REQUIRE( v.capacity() >= 5 );
    }
    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );
        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "reserving smaller does not change size or capacity" ) {
        v.reserve( 0 );
        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );
    }
}
```

For each `SECTION` the `TEST_CASE` is executed from the start. This means that each section is entered with a freshly constructed vector `v`, that we know has size 5 and capacity at least 5, because the two assertions are also checked before the section is entered. Each run through a test case will execute one, and only one, leaf section.

Section can also be nested, in which case the parent section can be entered multiple times, once for each leaf section. Nested sections are most useful when you have multiple tests that share part of the set up. To continue on the vector example above, you could add a check that `std::vector::reserve` does not remove unused excess capacity, like this:

```
    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );
        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );
        SECTION( "reserving down unused capacity does not change capacity" ) {
            v.reserve( 7 );
            REQUIRE( v.size() == 5 );
            REQUIRE( v.capacity() >= 10 );
        }
    }
```

Another way to look at sections is that they are a way to define a tree of paths through the test. Each section represents a node, and the final tree is walked in depth-first manner, with each path only visiting only one leaf node.

There is no practical limit on nesting sections, as long as your compiler can handle them, but keep in mind that overly nested sections can become unreadable. From experience, having section nest more than 3 levels is usually very hard to follow and not worth the removed duplication.

BDD style testing

Catch2 also provides some basic support for BDD-style testing. There are macro aliases for `TEST_CASE` and `SECTIONS` that you can use so that the resulting tests read as BDD spec. `SCENARIO` acts as a `TEST_CASE` with "Scenario: " name prefix. Then there are `GIVEN`, `WHEN`, `THEN` (and their variants with `AND_` prefix), which act as a `SECTION`, similarly prefixed with the macro name.

For more details on the macros look at the test cases and sections part of the reference docs,
or at the vector example done with BDD macros.

Data and Type driven tests

Test cases in Catch2 can also be driven by types, input data, or both at the same time.

For more details look into the Catch2 reference, either at the type parametrized test cases, or data generators.

Next steps

This page is a brief introduction to get you up and running with Catch2, and to show the basic features of Catch2. The features mentioned here can get you quite far, but there are many more. However, you can read about these as you go, in the ever-growing reference section of the documentation.