Tutorial

Contents

Getting Catch2

The simplest way to get Catch2 is to download the latest single header version. The single header is generated by merging a set of individual headers but it is still just normal source code in a header file.

Alternative ways of getting Catch2 include using your system package manager, or installing it using its CMake package.

The full source for Catch2, including test projects, documentation, and other things, is hosted on GitHub. http://catch-lib.net will redirect you there.

Where to put it?

Catch2 is header only. All you need to do is drop the file somewhere reachable from your project - either in some central location you can set your header search path to find, or directly into your project tree itself! This is a particularly good option for other Open-Source projects that want to use Catch for their test suite. See this blog entry for more on that.

The rest of this tutorial will assume that the Catch2 single-include header (or the include folder) is available unqualified - but you may need to prefix it with a folder name if necessary.

If you have installed Catch2 from system package manager, or CMake package, you need to include the header as `#include <catch2/catch.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;
}
```

To keep things simple we'll put everything in a single file (see later for more on how to structure your test files).

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cpp
file
#include 'catch.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').

If you run this as written it will pass. Everything is good. Right? Well, there is still a bug here. In fact the first version of this tutorial I posted here genuinely had the bug in! So it's not completely contrived (thanks to Daryle Walker (`@CTMacUser`) for pointing this out).

What is the bug? Well what is the factorial of zero? The factorial of zero is one - which is just one of those things you have to know (and remember!).

Let's add that 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 );
}
```

Now we get a failure - something like:

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

Note that we get the actual return value of Factorial(0) printed for us (0) - even though we used a natural expression with the == operator. That lets us immediately see what the problem is.

Let's change the factorial function to:

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

Now all the tests pass.

Of course there are still more issues to deal with. For example we'll hit problems when the return value starts to exceed the range of an unsigned int. With factorials that can happen quite quickly. You might want to add tests for such cases and decide how to handle them. We'll stop short of doing that here.

What did we do here?

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

1. All we did was `#define` one identifier and `#include` one header and we got everything - even an implementation of `main()` that will respond to command line arguments. You can only use that `#define` in one implementation file, for (hopefully) obvious reasons. Once you have more than one file with unit tests in you'll just `#include 'catch.hpp'` and go. Usually it's a good idea to have a dedicated implementation file that just has `#define CATCH_CONFIG_MAIN` and `#include 'catch.hpp'`. You can also provide your own implementation of main and drive Catch yourself (see Supplying-your-own-main()).

2. We introduce test cases with the `TEST_CASE` macro. This macro takes one or two arguments - a free form test name and, optionally, one or more tags (for more see Test cases and Sections). The test name must be unique. You can run sets of tests by specifying a wildcarded test name or a tag expression. See the command line docs for more information on running tests.
3. The name and tags arguments are just strings. We haven't had to declare a function or method - or explicitly register the test case anywhere. Behind the scenes a function with a generated name is defined for you, and automatically registered using static registry classes. By abstracting the function name away we can name our tests without the constraints of identifier names.
4. We write our individual test assertions using the `REQUIRE` macro. Rather than a separate macro for each type of condition we express the condition naturally using C/C++ syntax. Behind the scenes a simple set of expression templates captures the left-hand-side and right-hand-side of the expression so we can display the values in our test report. As we'll see later there *are* other assertion macros - but because of this technique the number of them is drastically reduced.

Test cases and sections

Most test frameworks have a class-based fixture mechanism. That is, test cases map to methods on a class and common setup and teardown can be performed in `setup()` and `teardown()` methods (or constructor/ destructor in languages, like C++, that support deterministic destruction).

While Catch fully supports this way of working there are a few problems with the approach. In particular the way your code must be split up, and the blunt granularity of it, may cause problems. You can only have one setup/ teardown pair across a set of methods, but sometimes you want slightly different setup in each method, or you may even want several levels of setup (a concept which we will clarify later on in this tutorial). It was problems like these that led James Newkirk, who led the team that built NUnit, to start again from scratch and build xUnit).

Catch takes a different approach (to both NUnit and xUnit) that is a more natural fit for C++ and the C family of languages. 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 - so as we enter each section we know that size is 5 and capacity is at least 5. We enforced those requirements with the `REQUIRE` s at the top level so we can be confident in them. This works because the `SECTION` macro contains an if statement that calls back into Catch to see if the section should be executed. One leaf section is executed on each run through a `TEST_CASE` . The other sections are skipped. Next time through the next section is executed, and so on until no new sections are encountered.

So far so good - this is already an improvement on the setup/teardown approach because now we see our setup code inline and use the stack.

The power of sections really shows, however, when we need to execute a sequence of checked operations. Continuing the vector example, we might want to verify that attempting to reserve a capacity smaller than the current capacity of the vector changes nothing. We can do that, naturally, like so:

```
    SECTION( 'reserving bigger changes capacity but not size' ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );

        SECTION( 'reserving smaller again does not change capacity' ) {
            v.reserve( 7 );

            REQUIRE( v.capacity() >= 10 );
        }
    }
```

Sections can be nested to an arbitrary depth (limited only by your stack size). Each leaf section (i.e. a section that contains no nested sections) will be executed exactly once, on a separate path of execution from any other leaf section (so no leaf section can interfere with another). A failure in a parent section will prevent nested sections from running - but then that's the idea.

BDD-Style

If you name your test cases and sections appropriately you can achieve a BDD-style specification structure. This became such a useful way of working that first class support has been added to Catch. Scenarios can be specified using `SCENARIO`, `GIVEN`, `WHEN` and `THEN` macros, which map on to `TEST_CASE`s and `SECTION`s, respectively. For more details see Test cases and sections.

The vector example can be adjusted to use these macros like so (example code):

```
SCENARIO( 'vectors can be sized and resized', '[vector]' ) {

    GIVEN( 'A vector with some items' ) {
        std::vector<int> v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( 'the size is increased' ) {
            v.resize( 10 );

            THEN( 'the size and capacity change' ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( 'the size is reduced' ) {
            v.resize( 0 );

            THEN( 'the size changes but not capacity' ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( 'more capacity is reserved' ) {
            v.reserve( 10 );

            THEN( 'the capacity changes but not the size' ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( 'less capacity is reserved' ) {
            v.reserve( 0 );

            THEN( 'neither size nor capacity are changed' ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Conveniently, these tests will be reported as follows when run:

```
Scenario: vectors can be sized and resized
    Given: A vector with some items
     When: more capacity is reserved
     Then: the capacity changes but not the size
```

Scaling up

To keep the tutorial simple we put all our code in a single file. This is fine to get started - and makes jumping into Catch even quicker and easier. As you write more real-world tests, though, this is not really the best approach.

The requirement is that the following block of code (or equivalent):

```
#define CATCH_CONFIG_MAIN
#include 'catch.hpp'
```

appears in *exactly one* source file. Use as many additional cpp files (or whatever you call your implementation files) as you need for your tests, partitioned however makes most sense for your way of working. Each additional file need only `#include 'catch.hpp'` - do not repeat the `#define`!

In fact it is usually a good idea to put the block with the `#define` in its own source file (code example main, tests).

Do not write your tests in header files!

Type parametrised test cases

Test cases in Catch2 can be also parametrised by type, via the `TEMPLATE_TEST_CASE` and `TEMPLATE_PRODUCT_TEST_CASE` macros, which behave in the same way the `TEST_CASE` macro, but are run for every type or type combination.

For more details, see our documentation on test cases and sections.

Next steps

This has been a brief introduction to get you up and running with Catch, and to point out some of the key differences between Catch and other frameworks you may already be familiar with. This will get you going quite far already and you are now in a position to dive in and write some tests.

Of course there is more to learn - most of which you should be able to page-fault in as you go. Please see the ever-growing Reference section for what's available.

Home