CATCHING UP WITH

Martin Hořeňovský @ Pex

CATCHING UP WITH

Martin Hořeňovský @ Pex

ABOUT THIS TALK

A new major release (v3) of Catch2 is coming,

A new major release (v3) of Catch2 is coming,

and as all major versions, that means changes.

For context, v3 has ~**500** comits extra over v2

For context, v3 has ~**500** comits extra over v2

there are **3786** commits total as of the time of writing

We will look into some changes that v3 will bring,

We will look into some changes that v3 will bring,

and also at some lesser known features in Catch2 v2.

CATCH2 ???

Catch2 is a fairly popular unit testing framework

Catch2 is a fairly popular unit testing framework

It differs significantly from xUnit frameworks

One of the big differences is in assertions.

One of the big differences is in assertions.

Assertions in Catch2 are just plain expressions:

One of the big differences is in assertions.

Assertions in Catch2 are just plain expressions:

```
   REQUIRE(factorial(0) == 1);
```

One of the big differences is in assertions.

Assertions in Catch2 are just plain expressions:

```
    REQUIRE(factorial(0) == 1);
```

```
C:\...\Catch2-talk-code\simple-usage.cpp(12): FAILED:
  REQUIRE( factorial(0) == 1 )
with expansion:
  0 == 1
```

The other big difference are sections.

The other big difference are sections.

Sections define multiple paths through test code, and are an effective replacement of fixtures for most cases.

The other big difference are sections.

Sections define multiple paths through test code, and are an effective replacement of fixtures for most cases.

```
TEST_CASE("Section showcase 1") {
    std::cout << '1';
    SECTION("A") {
        std::cout << 'A';
    }
    SECTION("B") {
        std::cout << 'B';
    }
    std::cout << '\n';
}
```

Sections can be nested arbitrarily deeply:

Sections can be nested arbitrarily deeply:

```
TEST_CASE("Section showcase 2") {
    std::cout << '1';
    SECTION("A") {
        std::cout << 'A';
        SECTION("a") { std::cout << 'a'; }
        SECTION("b") { std::cout << 'b'; }
    }
    SECTION("B") {
        std::cout << 'B';
        SECTION("a") { std::cout << 'a'; }
        SECTION("b") { std::cout << 'b'; }
    }
    std::cout << '\n';
}
```

And last, test case names are just strings.

And last, test case names are just strings.

```
TEST_CASE("Basic example") {

    REQUIRE(factorial(0) == 1);
}
```

CHANGES IN V3

THE BIG CHANGE

Catch2 is no longer distributed as a single header file,

Catch2 is no longer distributed as a single header file,

it is now a plain old static library with multiple headers.

WHY?

WHY?

- easier to maintain

WHY?

- easier to maintain
- cheaper to add new features

WHY?

- easier to maintain
- cheaper to add new features
- works better with dependency managers

WHY?

- easier to maintain
- cheaper to add new features
- works better with dependency managers
- compiles faster

EASIER TO MAINTAIN

EASIER TO MAINTAIN

The old single header distribution was lying.

EASIER TO MAINTAIN

The old single header distribution was lying.

There was one file with both a header and a cpp file.

EASIER TO MAINTAIN

The old single header distribution was lying.

There was one file with both a header and a cpp file.

The separation had to be maintained manually.

CHEAPER TO ADD NEW FEATURES

CHEAPER TO ADD NEW FEATURES

For single header, the bar to add features is high.

CHEAPER TO ADD NEW FEATURES

For single header, the bar to add features is high.

Everyone pays the compilation cost of new features.

CHEAPER TO ADD NEW FEATURES

For single header, the bar to add features is high.

Everyone pays the compilation cost of new features.

For context, the header has **642 KB** and **18k** lines.

WORKS BETTER WITH DEP. MANAGERS

WORKS BETTER WITH DEP. MANAGERS

With single header distribution, the user still has to provide an "implementation" TU.

WORKS BETTER WITH DEP. MANAGERS

With single header distribution, the user still has to provide an "implementation" TU.

With a classic library the user can just include headers.

COMPILES FASTER

COMPILES FASTER

Less code compiles faster.

COMPILES FASTER

Less code compiles faster.

Who knew? ¯_(ツ)_/¯

WHY NOT?

WHY NOT?

- If you manage your dependencies poorly, you can have issues with inconsistent compilation options

WHY NOT?

- If you manage your dependencies poorly, you can have issues with inconsistent compilation options
- Harder* to vendor into your own project

OTHER CHANGES IN V3

C++14 is the minimum supported language version

COMPILATION TIMES

COMPILATION TIMES

file contents	speedup over v2
include	1.76x*
100 tests	1.36x
100 tests, 5 sections	1.10x

* About 160ms

RUNTIME PERFORMANCE

RUNTIME PERFORMANCE

task	debug	release
Check 1M assertions	1.10	1.02
Run 100 tests, 9 leaf sections	1.27	1.04
Run 3k tests	1.49	1.22
Run 1 out of 3k tests	1.79	2.06

GENERIC MATCHERS

GENERIC MATCHERS

Matchers have been around since "Catch Classic" (1.x).

GENERIC MATCHERS

Matchers have been around since "Catch Classic" (1.x).

They encapsulate a predicate on expected test output.

GENERIC MATCHERS

Matchers have been around since "Catch Classic" (1.x).

They encapsulate a predicate on expected test output.

Old matchers can only match concrete types.

Generic matchers can provide overloaded and templated `match` member function as needed.

Generic matchers can provide overloaded and templated `match` member function as needed.

```
template <typename Range>
struct EqualsRangeMatcher : Catch::Matchers::MatcherGenericBase {

    // ... constructors, etc ...

    template <typename OtherRange>
    bool match(OtherRange const& other) const {
        using std::begin; using std::end;

        return std::equal(
            begin(m_range), end(m_range),
            begin(other), end(other));
    }
};
```

Catch2 provides some generic matchers built in:

Catch2 provides some generic matchers built in:

- `IsEmpty`

Catch2 provides some generic matchers built in:

- `IsEmpty`
- `SizeIs`

Catch2 provides some generic matchers built in:

- `IsEmpty`
- `SizeIs`
- `Contains`

Catch2 provides some generic matchers built in:

- `IsEmpty`
- `SizeIs`
- `Contains`
- `AllMatch`, `AnyMatch`, `NoneMatch`

Catch2 provides some generic matchers built in:

- `IsEmpty`
- `SizeIs`
- `Contains`
- `AllMatch`, `AnyMatch`, `NoneMatch`

More will be added over time

REPORTER CHANGES

REPORTER CHANGES

Reporters are a customization point for testing output

REPORTER CHANGES

Reporters are a customization point for testing output

They decide how tests, assertions, etc, are reported

REPORTER CHANGES

Reporters are a customization point for testing output

They decide how tests, assertions, etc, are reported

v3 allows for multiple reporters to be active

Each reporter can write to different file, or to stdout

Each reporter can write to different file, or to stdout

```
./tests/SelfTest -r junit:junit.xml -r console
```

Each reporter can write to different file, or to stdout

```
./tests/SelfTest -r junit:junit.xml -r console
```

This writes the JUnit XML file to a file, and prints the user-friendly console output to stdout

Multireporters make partial reporters useful, e.g. for writing out benchmark results into markdown tables.

v3 also made listings customizable by reporters

v3 also made listings customizable by reporters

e.g. the XML reporter outputs XML listings

```
$ ./tests/SelfTest -r xml --list-tests
<?xml version="1.0" encoding="UTF-8"?>
<MatchingTests>
  <TestCase>
    <Name>Test with special, characters "in name</Name>
    <ClassName/>
    <Tags>[cli][regression]</Tags>
    <SourceInfo>
      <File>/.../CmdLine.tests.cpp</File>
      <Line>574</Line>
    </SourceInfo>
  </TestCase>
  ...
```

Reporters actually went through bunch more changes:

Reporters actually went through bunch more changes:

- Redundant reporter events were removed

Reporters actually went through bunch more changes:

- Redundant reporter events were removed
- New useful reporter events were added

Reporters actually went through bunch more changes:

- Redundant reporter events were removed
- New useful reporter events were added
- Some existing events had their API changed

Reporters actually went through bunch more changes:

- Redundant reporter events were removed
- New useful reporter events were added
- Some existing events had their API changed
- Reporter bases were refactored

RECAP

RECAP

- v3 brings better compile and runtime performance

RECAP

- v3 brings better compile and runtime performance
- Reporters will become a lot more powerful

RECAP

- v3 brings better compile and runtime performance
- Reporters will become a lot more powerful
- Matchers can be written to be much more generic

LESS USED FEATURES

We will look at:

We will look at:

- Data driven tests (generators)

We will look at:

- Data driven tests (generators)
- Type driven tests (templated test cases)

We will look at:

- Data driven tests (generators)
- Type driven tests (templated test cases)
- Listeners

We will look at:

- Data driven tests (generators)
- Type driven tests (templated test cases)
- Listeners
- Micro benchmarking support

DATA DRIVEN TESTS

Data driven testing means using the same test code for different inputs.

Data driven testing means using the same test code for different inputs.

(De)Serializing types is a common example.

```
WaitForKeypress::When toWaitForKeypress(std::string const& input)
```

```
WaitForKeypress::When toWaitForKeypress(std::string const& input)
```

```
TEST_CASE("WaitForKeypress parsing") {
    auto [input, output] = GENERATE(
        table<char const*, WaitForKeypress::When>({
            {"never", WaitForKeypress::Never},
            {"start", WaitForKeypress::BeforeStart},
            {"exit",  WaitForKeypress::BeforeExit},
            {"both",  WaitForKeypress::BeforeStartAndExit},
        })
    );

    REQUIRE(toWaitForKeypress(input) == output);
}
```

There can be multiple GENERATEs per test/section.

There can be multiple GENERATEs per test/section.

They produce a cartesian product of all values.

There can be multiple GENERATEs per test/section.

They produce a cartesian product of all values.

```
TEST_CASE("Exhaustive config check") {
    bool implicative_blocks = GENERATE(true, false);
    bool implicative_diffs = GENERATE(true, false);

    // Actual test...
}
```

Test-path-wise, `GENERATE` behaves like a `SECTION`

Test-path-wise, GENERATE behaves like a SECTION

```
TEST_CASE("Nesting generators with SECTIONs") {
    auto number = GENERATE(2, 4);
    SECTION("A") {
        std::cout << "A\n";
    }
    SECTION("B") {
        auto number2 = GENERATE(1, 3);
        std::cout << "B\n";
    }
}
```

Test-path-wise, GENERATE behaves like a SECTION

```
TEST_CASE("Nesting generators with SECTIONs") {
    auto number = GENERATE(2, 4);
    SECTION("A") {
        std::cout << "A\n";
    }
    SECTION("B") {
        auto number2 = GENERATE(1, 3);
        std::cout << "B\n";
    }
}
```

Prints out "A\n", "B\n", "B\n", "A\n", "B\n", "B\n"

Generators do not have to know their size up front.

Generators do not have to know their size up front.

They can even be infinite, relying on later termination.

Generators do not have to know their size up front.

They can even be infinite, relying on later termination.

```
auto i = GENERATE(take(10,
                  random(-100., 100.))
);
```

You can also mix constants and complex generators.

You can also mix constants and complex generators.

```
TEST_CASE("Mixing literals and complex generators") {
    auto i = GENERATE(88.2,
                      take(10,
                      random(-100., 100.))
    );
    // test code
}
```

In v3 GENERATE decays literal arguments.

In v3 GENERATE decays literal arguments.

This fixes a possible bug when mixing primitive type literals and complex generators.

In v3 GENERATE decays literal arguments.

This fixes a possible bug when mixing primitive type literals and complex generators.

Also fixes mixing different length string literals.

You can of course write your own generators.

You can of course write your own generators.

For details look into the documentation.

TYPE DRIVEN TESTS

Type driven testing means using same test code for different types.

Type driven testing means using same test code for different types.

This is useful for testing generic code (e.g. containers).

```
TEMPLATE_TEST_CASE("You can have a test across multiple types", "",
                   int, float) {
    std::vector<TestType> vec;

    vec.reserve(5);
    REQUIRE(vec.size() == 0);
    REQUIRE(vec.capacity() >= 5);
}
```

Catch2 provides a lot of macros for templated tests:

- TEMPLATE_TEST_CASE
- TEMPLATE_LIST_TEST_CASE
- TEMPLATE_PRODUCT_TEST_CASE
- TEMPLATE_TEST_CASE_SIG
- and some more

Due to preprocessor limitations, types with commas (e.g. `map<string, string>`) need to be passed in parentheses (e.g. `(map<string, string>)`).

LISTENERS

Listeners are reporters without output.

Listeners are reporters without output.

Instead they perform actions within the test process.

Listeners are reporters without output.

Instead they perform actions within the test process.

e.g. they can initialize a C library before testing starts

Listeners are reporters without output.

Instead they perform actions within the test process.

e.g. they can initialize a C library before testing starts

... or they can drop logs if the test case passed.

```
class testRunListener : public Catch::EventListenerBase {
public:
    using Catch::EventListenerBase::EventListenerBase;

    void testRunStarting(Catch::TestRunInfo const&) override {
        lib_foo_init();
    }
};

CATCH_REGISTER_LISTENER(testRunListener)
```

There are 21 different listener/reporter events.

There are 21 different listener/reporter events.

- 4 for benchmarking

There are 21 different listener/reporter events.

- 4 for benchmarking
- 3 for listing things

There are 21 different listener/reporter events.

- 4 for benchmarking
- 3 for listing things
- 10 (5 pairs) for running tests

There are 21 different listener/reporter events.

- 4 for benchmarking
- 3 for listing things
- 10 (5 pairs) for running tests
- 4 miscellaneous

(MICRO)BENCHMARKING

Catch2 contains adapted code from Nonius.

Catch2 contains adapted code from Nonius.

Benchmarking blocks are written inside tests.

Benchmark block is started with a BENCHMARK macro

Benchmark block is started with a BENCHMARK macro

```
BENCHMARK("factorial 30") {
    return factorial(30); // <-- won't be optimized away
}; // <-- The semicolon must be there
```

```
$ ./benchmarks "Simple benchmark"
-------------------------------------------------------------------
Simple benchmark
-------------------------------------------------------------------
C:\ubuntu\presentations\Catch2-talk-examples\benchmarks.cpp(7)
...................................................................

benchmark name                samples       iterations    estimated
                              mean          low mean      high mean
                              std dev       low std dev   high std dev
-------------------------------------------------------------------
factorial 30                        100         56111     5.6111 ms
                             0.806206 ns    0.80412 ns   0.810233 ns
                            0.0142946 ns 0.00862981 ns  0.0224988 ns
```

You can use generators to run benchmarks across different inputs:

You can use generators to run benchmarks across different inputs:

```
TEST_CASE("Parametrized benchmark") {
    auto input = GENERATE(1, 10, 15, 20, 30);

    BENCHMARK("factorial " + std::to_string(input)) {
        return factorial(input);
    };
}
```

The code inside the benchmark block will be executed multiple times.

The code inside the benchmark block will be executed multiple times.

It **must** be meaningfully repeatable.

You *can* have assertions inside benchmark block.

You *can* have assertions inside benchmark block.

```
    BENCHMARK("require(true)") {
        REQUIRE(true);
    };
```

You *can* have assertions inside benchmark block.

```
BENCHMARK("require(true)") {
    REQUIRE(true);
};
```

But they will be counted in the measured time.

There are more benchmarking utilities in Catch2,

There are more benchmarking utilities in Catch2,

you can find them in the benchmarking docs.

RECAP

v3 is coming soon (?) and targets C++14

v3 is coming soon (?) and targets C++14

v3 will be statically compiled library

v3 is coming soon (?) and targets C++14

v3 will be statically compiled library

v3 brings performance improvements

v3 is coming soon (?) and targets C++14

v3 will be statically compiled library

v3 brings performance improvements

v3 brings some useful new features

Catch2 already supports

Catch2 already supports

- non-generic matchers

Catch2 already supports

- non-generic matchers
- benchmarking

Catch2 already supports

- non-generic matchers
- benchmarking
- custom reporters

Catch2 already supports

- non-generic matchers
- benchmarking
- custom reporters
- event listeners

Catch2 already supports

- non-generic matchers
- benchmarking
- custom reporters
- event listeners
- data and type parametrized tests

Catch2 already supports

- non-generic matchers
- benchmarking
- custom reporters
- event listeners
- data and type parametrized tests
- and more...

Catch2 already supports

- non-generic matchers
- benchmarking
- custom reporters
- event listeners
- data and type parametrized tests
- and more...

Catch2 usually provides good docs, read them!

THE END

QUESTIONS?

- https://github.com/horenmar/Catch2-talk-examples