C++ Unit Testing using Catch

Nikola Ajzenhamer

Overview

- Module 1: Introducing Catch
  - What is Catch and unit testing
  - Setting up Catch
- Module 2: Organizing tests using Catch
  - Naming tests and using tags
  - Using Catch from the command line
- Module 3: Asserting results using Catch
  - Using REQUIRE and ASSUME
  - Checking for exceptions
  - Getting detailed information from tests
- Module 4: Reducing duplicate test code
  - Using test fixtures vs. sections
  - Writing BDD style tests

Module 1: Introducing Catch

Unit test

- A method (code)
- Tests specific functionality
- Clear pass/fail criteria
- Runs in isolation

Simple unit test (example)

```
TEST_CLASS(MyUnitTest)
{
    public:
        TEST_METHOD(TestMethod1)
        {
            // Your test code here
        }
}
```

- TEST_CLASS – defines a class which has all the tests inside
- TEST_METHOD – defines one specific test
    - Names for both of these are given as an argument to the macro

Why write automated tests?

- Quick feedback
  - Finding bugs quickly, without needing to wait
  - Shows whether or not we wrote the right code
- Avoid stupid bugs
  - Finding nullptr, invalid indexes, etc.
- Immune to Regression
  - Introducing a bug will fail another unit test
- Change your code without fear
- In code documentation
- You are already testing your code, so why not automating it?

Catch

- C++ Automated Cases in Headers
- Open source (https://github.com/philsquared/Catch)

Why use Catch?

- Single header deployment
- No external dependencies
- Tests’ names are free-form strings
- Powerful “Assertions”
- Excellent error messages
- Sections
    - Feature that other unit test frameworks does not have

Getting started with Catch

- Download catch.hpp
- #define CATCH_CONFIG_MAIN
  - Instead of the int main(int argc, char** argv) method
- Include “catch.hpp”

```
TEST_CASE(“This is a test name”)
{
    // …
}
```

Writing tests using Catch (example)

```
TEST_CASE(“This is a test name”, “[Tag]”)
{
    MyClass myClass;

    REQUIRE(myClass.MeaningOfLife() == 42);
}
```

- TEST_CASE – declared a new test case
- REQUIRE – used for asserting the result of the test

T9 Predictive Text Algorithm

- Unit testing T9 algorithm
  - Used in older cell phones
  - Input: a sequence of digits
  - Output: suggested words
- Examples:
  - 843 → “the”
  - 4663 → “good”
  - 43556 → “hello”
- Naïve implementation

Test 1 for T9 algorithm (example)

```
#define CATCH_CONFIG_MAIN

#include "catch.hpp"

#include "T9Predict/Engine.h"
#include "T9Predict/WordsTree.h"

TEST_CASE("Called with empty digit list --> returns no results")
{
    WordsTree emptyTree;
    Engine t9Engine(emptyTree);

    Digits digits;

    auto result = t9Engine.GetWordsByDigits(digits);

    REQUIRE(result.size() == 0);
}
```

Parts of the test case

- Arrange
  - Used for setting up everything we need for the test
- Act
  - Running the actual test
- Assert
  - Checking the results of the test

Test 1 for T9 algorithm (example)

```
#define CATCH_CONFIG_MAIN

#include "catch.hpp"

#include "T9Predict/Engine.h"
#include "T9Predict/WordsTree.h"

TEST_CASE("Called with empty digit list --> returns no results")
{
    WordsTree emptyTree;
    Engine t9Engine(emptyTree);

    Digits digits;

    auto result = t9Engine.GetWordsByDigits(digits);

    REQUIRE(result.size() == 0);
}
```

Arrange

Act

Assert

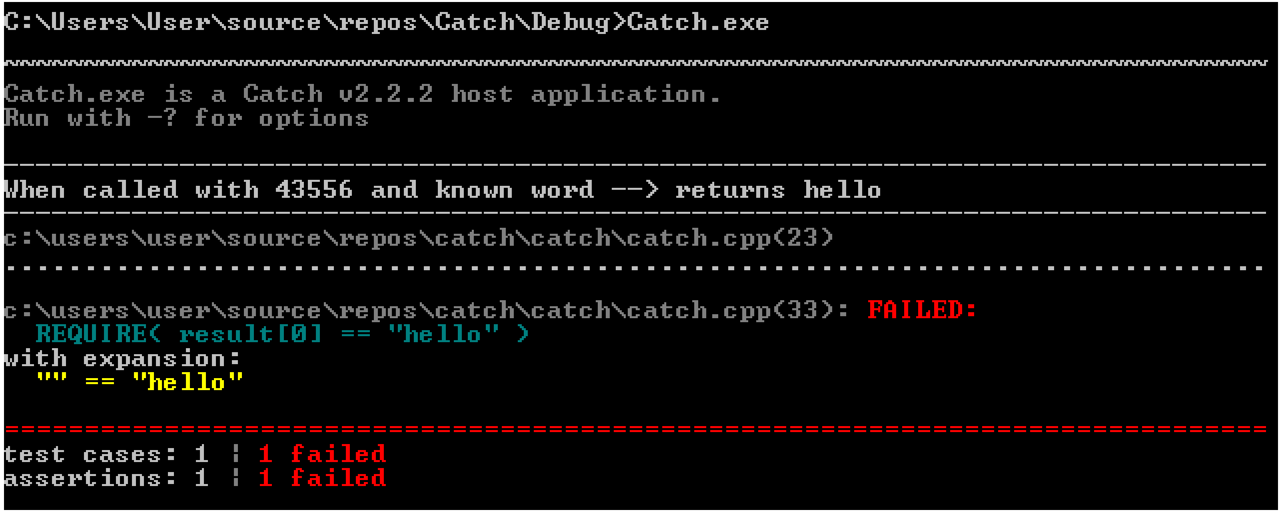
Running the Test 1 for T9 algorithm

```
C:\Users\User\source\repos\Catch\Debug>Catch.exe
===============================================================================
All tests passed (1 assertion in 1 test case)
```

Test 2 for T9 algorithm (example)

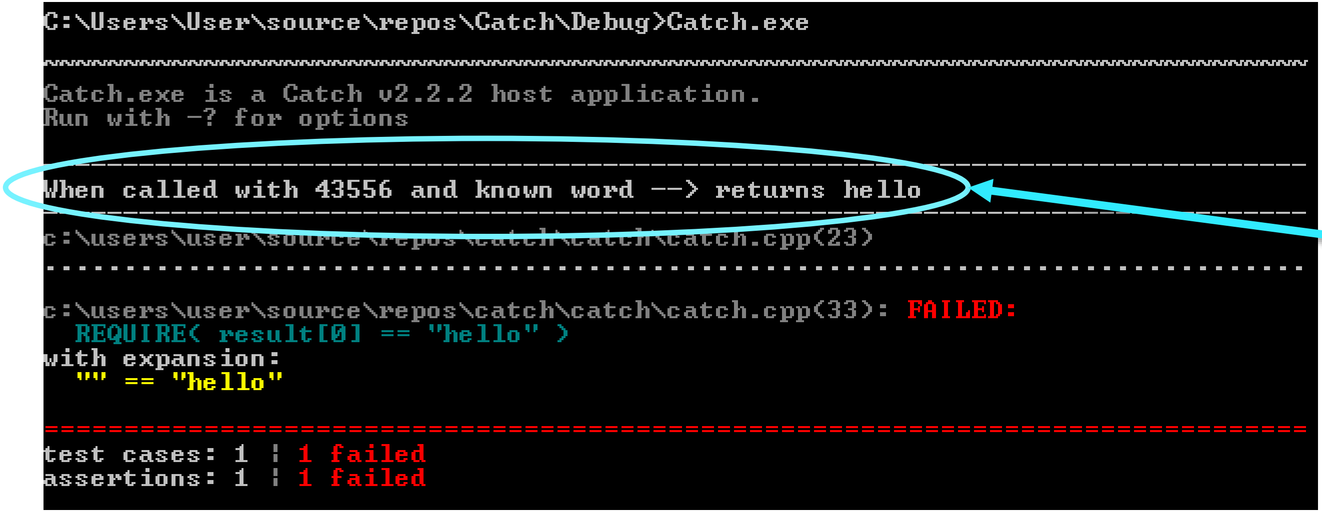
```
#define CATCH_CONFIG_MAIN

#include "catch.hpp"

#include "T9Predict/Engine.h"
#include "T9Predict/WordsTree.h"

TEST_CASE("When called with 43556 and known word --> returns hello")
{
    WordsTree knownWords;
    knownWords.AddWord("hello");
    Engine t9Engine(knownWords);

    Digits digits = { 4, 3, 5, 5, 6 };

    auto result = t9Engine.GetWordsByDigits(digits);

    REQUIRE(result[0] == "hello");
}
```

Running the Test 2 for T9 algorithm



```
C:\Users\User\source\repos\Catch\Debug>Catch.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Catch.exe is a Catch v2.2.2 host application.
Run with -? for options

-------------------------------------------------------------------------------
When called with 43556 and known word --> returns hello
-------------------------------------------------------------------------------
c:\users\user\source\repos\catch\catch\catch.cpp(23)
...............................................................................

c:\users\user\source\repos\catch\catch\catch.cpp(33): FAILED:
  REQUIRE( result[0] == "hello" )
with expansion:
  "" == "hello"

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

Running the Test 2 for T9 algorithm



```
C:\Users\User\source\repos\Catch\Debug>Catch.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Catch.exe is a Catch v2.2.2 host application.
Run with -? for options

-------------------------------------------------------------------------------
When called with 43556 and known word --> returns hello
-------------------------------------------------------------------------------
c:\users\user\source\repos\catch\catch\catch.cpp(23)
...............................................................................

c:\users\user\source\repos\catch\catch\catch.cpp(33): FAILED:
  REQUIRE( result[0] == "hello" )
with expansion:
  "" == "hello"

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

Name of the test that failed

Running the Test 2 for T9 algorithm

```
C:\Users\User\source\repos\Catch\Debug>Catch.exe

Catch.exe is a Catch v2.2.2 host application.
Run with -? for options

-------------------------------------------------------------------------------
When called with 43556 and known word --> returns hello
-------------------------------------------------------------------------------
c:\users\user\source\repos\catch\catch\catch.cpp(23)
...............................................................................

c:\users\user\source\repos\catch\catch\catch.cpp(33): FAILED:
  REQUIRE( result[0] == "hello" )
with expansion:
  "" == "hello"

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

Name of the test that failed

Code that failed

Running the Test 2 for T9 algorithm

```
C:\Users\User\source\repos\Catch\Debug>Catch.exe

Catch.exe is a Catch v2.2.2 host application.
Run with -? for options

-------------------------------------------------------------------------------
When called with 43556 and known word --> returns hello
-------------------------------------------------------------------------------
c:\users\user\source\repos\catch\catch\catch.cpp(23)
...............................................................................

c:\users\user\source\repos\catch\catch\catch.cpp(33): FAILED:
  REQUIRE( result[0] == "hello" )
with expansion:
  "" == "hello"

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

Name of the test that failed

Code that failed

Result

Comparing Catch to Traditional frameworks (xUnit)

```
TEST_METHOD(PassDigitsForHelloReturnCorrectString) {
    WordsTree emptyTree;
    Engine t9Engine(emptyTree);

    Digits digits = {4, 3, 5, 5, 6};
    auto result = t9Engine.GetWordsByDigits(digits);

    Assert::AreEqual(std::string(“hello”), result[0]);
}
```

Comparing Catch to Traditional frameworks (xUnit)

```
TEST_METHOD(PassDigitsForHelloReturnCorrectString) {
    WordsTree emptyTree;
    Engine t9Engine(emptyTree);

    Digits digits = {4, 3, 5, 5, 6};
    auto result = t9Engine.GetWordsByDigits(digits);

    Assert::AreEqual(std::string("hello"), result[0]);
}
```

The test name must be a valid method name

Comparing Catch to Traditional frameworks (xUnit)

```
TEST_METHOD(PassDigitsForHelloReturnCorrectString) {
    WordsTree emptyTree;
    Engine t9Engine(emptyTree);

    Digits digits = {4, 3, 5, 5, 6};
    auto result = t9Engine.GetWordsByDigits(digits);

    Assert::AreEqual(std::string("hello"), result[0]);
}
```

Several types of asserting for various outcomes

Comparing Catch to Traditional frameworks (xUnit)

```
TEST_METHOD(PassDigitsForHelloReturnCorrectString) {
    WordsTree emptyTree;
    Engine t9Engine(emptyTree);

    Digits digits = {4, 3, 5, 5, 6};
    auto result = t9Engine.GetWordsByDigits(digits);

    Assert::AreEqual(std::string(“hello”), result[0]);
}
```

Two arguments – the first is the expected value, and the second is the actual value

Comparing Catch to Traditional frameworks (xUnit)

```
TEST_METHO
    WordsT
    Engine

    Digits
    auto r

    Assert
}
```

PassDigitsReturnOneString

Source: unittest1.cpp line 26

Test Failed - PassDigitsReturnOneString

Message: Assert failed. Expected:<hello> Actual:<>

Elapsed time: 4 ms

StackTrace:

T9EngineTests::PassDigitsReturnOneString()

Comparing Catch to Traditional frameworks (xUnit)

- Traditional (xUnit)
  - Names must be valid method names
  - Several methods (Assert class)
  - Failure messages depends on assertion
- Catch
  - Names are strings
  - One REQUIRE
  - Out of the box detailed failure messages

Module 2: Organizing tests using Catch

Why do we need a good test name?

- Understand what the code does
- Execute a subset of your tests
  - Using the Catch command line
- Find the test you need
- Failures
  - The first thing we see when the test fails is the name
  - Root cause analysis
- A good test name will help you write a good test

Avoid at all costs

- “Test 1_11_37”
  - Completely unreadable
  - Even if I do remember it – it requires a lot of effort to be remembered
- “Customer Test Simple”
  - A little better – we know it is testing a Customer
  - Still not sure what are we testing it for or what we expect it to do
  - Too generic
- “WorkItem1234”
  - Requires searching for the item number 1234 – too cumbersome
- “Workload error exception”
  - Not enough information

Rules for good names

- Express a specific requirement or behavior
- Should include:
  1. The starting state (what we are testing)
  2. Given input (what action we perform on the system)
  3. Expected result
  - Unless some of the three above are irrelevant
- Should be easily found
- Should not include “Test”

Good names (examples)

- If Age > 18 IsAdult returns true
  - It is clear to see what is being tested
- When called with xyz Then return true
  - We use ‘When” to emphasize the action taken (“called”) with the specific value
- Should throw exception When invalid user
  - Starting from the result (“Should [do something] When [condition happens]”)
- Given authenticated When invalid number used Then transaction will fail
  - Given [some state] When [something happens] Then [do something]
  - Common in BDD frameworks
- Called with empty list → return nullptr
  - Inventing our own
  - → displays the connection between the cause and the consequence

Catch & the command line

- Catch tests are compiled into a console app (exe)
  - Running the executable runs all the tests
- Use arguments to specify which test(s) to run
  - Running the tests on the subset of the system
  - Build servers
    - Which tests run with every commit
    - Which tests run once a day (week, month, etc.)

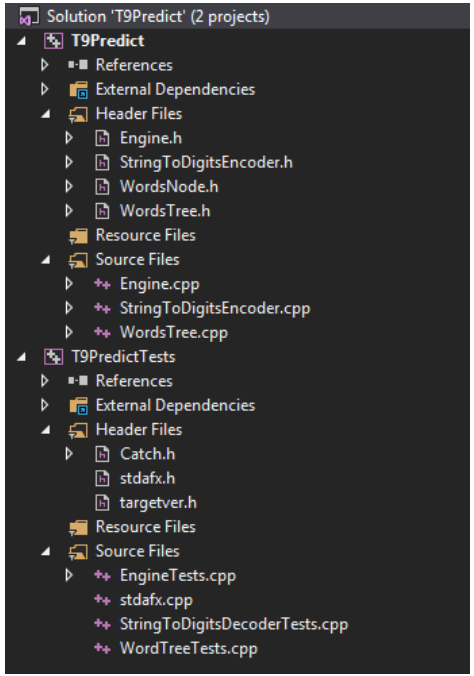
Test name related arguments

- Run specific test
  - `testName`
  - `“Test Name”`
- Run group of tests
  - Using the `*` character to specify a wildcard, f.e. `Customer*`, `*Customer`, `*Customer*`
- Run all tests except
  - `Exclude:testName` or `~testName`
- Combination of all the previous

Assumptions for further analysis

- There are more code implementations in the T9Predict algorithm project
- There are more test cases written
- The application that contains these test cases is called `T9PredictTests.exe`
- Test are split into 3 source files in T9PredictTests project
  - Every testing source is named after the source (essentially, the class) that it tests
  - For example, `EngineTests.cpp` is used to test `Engine.cpp`

- Solution explorer



Running all the tests

- Simply calling the application

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe
===============================================================================
All tests passed (74 assertions in 12 test cases)
```

Listing all the test without running them

- Use the option `-l` or `--list-tests`

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe -l
All available test cases:
  Encode single uppercase letter --> return correct digit
      [Single Digit]
  Encode single lowercase letter --> return correct digit
      [Single Digit]
  Given a string Hello return 43556
      [End2End][Regression]
  Called with empty digit list --> returns no results
      [Empty][Init]
  When called with 43556 and known word then return hello
      [End2End][Search]
  When called with 4663 and known words then return these words
      [End2End][Search][multiple words]
  When called with 4663 and known words but other words exist then return only
    these words
      [End2End][Search][multiple words]
  If no words exist GetWords returns empty collection
      [Init]
  Find one letter word which exist in tree
      [Search]
  Find two one letter words which exist in tree
      [Search]
  Find two letter word which exist in tree
      [Search]
  Find word when tree has different encoded word begins with same letter
      [Search]
12 test cases
```

Listing only the names, without tags

- Use the option `--list-test-names-only`

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe --list-test-names-
only
Encode single uppercase letter --> return correct digit
Encode single lowercase letter --> return correct digit
Given a string Hello return 43556
Called with empty digit list --> returns no results
When called with 43556 and known word then return hello
When called with 4663 and known words then return these words
When called with 4663 and known words but other words exist then return only the
se words
If no words exist GetWords returns empty collection
Find one letter word which exist in tree
Find two one letter words which exist in tree
Find two letter word which exist in tree
Find word when tree has different encoded word begins with same letter
```

Running only the argument specified tests (example)

- Running all the tests that begin with `Find`

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe Find*
===============================================================================
All tests passed (9 assertions in 4 test cases)
```

- Running all the tests that include the “`correct digit`” clause

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe "*correct digit*"
===============================================================================
All tests passed (52 assertions in 2 test cases)
```

Tags

- A simple way to group tests
- Additional strings associated with test case
  - Must begin with an alphanumeric character
- Tag names are not case sensitive

```
TEST_CASE(“This is a test case name”, “[tag1][tag2]”)
{
    // …
}
```

Usage of tags

- Categorization of tests
  - Form of documentation
  - Practical way to only execute a subset of tests
- Types of categorizations
  - By code under test – testing components (DB, Model, Customer)
  - By test type (Unit, Integration, Scenario)
  - By execution speed (Slow, Fast)
  - All of the above, and many more, fall under user-defined categories

Running tests by tags

- The same as with names,
  except we put them in square brackets
- Additional cases
  - Running tests that have multiple tags (AND relationship)

    “`[tag1][tag2]…[tagN]`”
  - Running tests that have at least one of the tags (OR relationship)

    “`[tag1],[tag2],…,[tagN]`”

Special tags

Tag	Explanation	Examples of usage / Comments
`[!hide]` or `[.]`	Declares that the test should be skipped/ignored	`[.][tagName]` or `[.tagName]` results in skipping the tag `tagName`
`[!throws]`	Marks the test as one that might throw an exception	Test will not run if we run the test executable with option `-e` or `--nothrow`
`[!shouldfail]`	Specifies that the test is supposed to fail	Reverses the passing logic – pass if fail, and fail if pass
`[!mayfail]`	Does not fail the test if assertion fails	For defining requirements, which are not yet implemented
`[#<filename>]`	Runs only the tests in the `<filename>`	For command line usage only. Use option `-#` to specify `<filename>` as tag

Listing all the tests' tags

- Use the option `-t` or `--list-tags`

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe --list-tags
All available tags:
   1  [Empty]
   4  [End2End]
   2  [Init]
   2  [multiple words]
   1  [Regression]
   7  [Search]
   2  [Single Digit]
7 tags
```

Listing all the tests which have certain tag

- Use the option `-l` along with the tag

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe -l [End2End]
Matching test cases:
  Given a string Hello return 43556
      [End2End][Regression]
  When called with 43556 and known word then return hello
      [End2End][Search]
  When called with 4663 and known words then return these words
      [End2End][Search][multiple words]
  When called with 4663 and known words but other words exist then return only
    these words
      [End2End][Search][multiple words]
4 matching test cases
```

Listing all the tests with the filenames they are in

- Use the options `-l -#`

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe -l -#
All available test cases:
  Encode single uppercase letter --> return correct digit
      [#stringtodigitsdecodertests][Single Digit]
  Encode single lowercase letter --> return correct digit
      [#stringtodigitsdecodertests][Single Digit]
  Given a string Hello return 43556
      [#stringtodigitsdecodertests][End2End][Regression]
  Called with empty digit list --> returns no results
      [#enginetests][Empty][Init]
  When called with 43556 and known word then return hello
      [#enginetests][End2End][Search]
  When called with 4663 and known words then return these words
      [#enginetests][End2End][Search][multiple words]
  When called with 4663 and known words but other words exist then return only
    these words
      [#enginetests][End2End][Search][multiple words]
  If no words exist GetWords returns empty collection
      [#wordtreetests][Init]
  Find one letter word which exist in tree
      [#wordtreetests][Search]
  Find two one letter words which exist in tree
      [#wordtreetests][Search]
  Find two letter word which exist in tree
      [#wordtreetests][Search]
  Find word when tree has different encoded word begins with same letter
      [#wordtreetests][Search]
12 test cases
```

Listing the tags and filenames

- Use the options `-t -#`
- Useful for seeing how many tests are in each filename

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe -t -#
All available tags:
   4  [#enginetests]
   3  [#stringtodigitsdecodertests]
   5  [#wordtreetests]
   1  [Empty]
   4  [End2End]
   2  [Init]
   2  [multiple words]
   1  [Regression]
   7  [Search]
   2  [Single Digit]
10 tags
```

Running all the tests from a file

- Use the option `-#` along with the `[#<filename>]`

```
C:\Users\User\source\repos\Catch\x64\Debug>T9PredictTests.exe -# [#enginetests]
===============================================================================
All tests passed (11 assertions in 4 test cases)
```

Tag aliases

- Combination of tags and rules with them
  can be bothersome to remember and type every time
- We can create an alias for such combinations
- Aliases start with @ symbol

```
CATCH_REGISTER_TAG_ALIAS(“[@abc]”, “[a],[b]~[c]”)
```

More command line arguments

Option	Explanation
-h, -?, --help	Shows a list of all the available commands and their descriptions
-f, --input_file <filename>	Defines a file with a list of tests to run
-o, --out <filename>	Redirects a Catch output to a file
-b, --break	Each time a test fails, Catch will break the debugger at that specific point. Not all IDEs support this option (Visual Studio and Xcode do)
-a, --abort	Tells the Catch to stop the testing on the first failure
-x, --abortx [<threshold>]	Tells the Catch to stop the testing after the <threshold> number of failed tests

Module 3: Asserting results using Catch

REQUIRE

- Single macro for all/most assertion needs
- Write the assertion in plain code
- Excellent failure messages

Why should you care about failure messages?

- Understand why the test failed
  - Helps in fixing the problem
- Reduce debugging time
- It’s the purpose of the test
  - We don’t write tests we know it will pass
  - In other words, we care about the failures

What is 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);
}
```

Is it okay to use multiple assertions in one test?

- The singular assert per test rule
  - Each test has a singular assert
- If that is not the case,
  split the test into multiple tests

The problem with multiple assertions

- Lose information
  - When an assert fails, it throws an exception
  - If the assert *i* fails, then we lose all the information about all the asserts *i+1*, *i+2*, etc.
- Testing more than one aspect
  - Every test should focus on one and only one scenario
- 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");
}
```

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");
}
```

Checks if the result has only one value

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");
}
```

Checks if that one value is the correct one

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");
}
```

We are testing two aspects of the same results, not two different aspects!

When to use multiple assertions?

- Multiple checks for single “concept”
  - We will see later how to handle the exceptions at the *i*-th failure
- Checking related logic
- Always be pragmatic
  - One assert per test is a good rule of the thumb
  - With experience comes better reasoning

CHECK

- A macro that works like a REQUIRE
  with the exception that it does not abort the test
  until the test is finished

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

Asserting a false result

- REQUIRE_FALSE, CHECK_FALSE
- Work exactly like the “positive” counterparts, with the exception of checking for false condition
- Useful in cases where an ! operator would be used:
  - Instead of

```
REQUIRE(!MethodReturnsFalse()); // will not compile
```

Asserting a false result

- REQUIRE_FALSE, CHECK_FALSE
- Work exactly like the “positive” counterparts,
  with the exception of checking for false condition
- Useful in cases where an ! operator would be used:
  - Instead of

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

    Use

    ```
    REQUIRE_FALSE(MethodReturnsFalse()); // correct version, does the same
    ```

Multiple assertions in one test (example)

- Does this test have problems?

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;

    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);

    tree.AddWord(“a”, { 2 });

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

    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == “a”);
}
```

Multiple assertions in one test (example)

- Problem 1: The name does not correspond to the test code

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;

    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);

    tree.AddWord(“a”, { 2 });

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

    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == “a”);
}
```

Multiple assertions in one test (example)

- Problem 2: This test does not conform to the every-test-has-three-parts rule

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;

    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);

    tree.AddWord(“a”, { 2 });

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

    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == “a”);
}
```

Arrange

Act + Assert?

Arrange again?

Act again?

Assert again?

Multiple assertions in one test (example)

- First step in solution: split into two tests

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;
    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);
}

TEST_CASE("Find one letter word which exist in tree", "[Search]")
{
    WordsTree tree;
    tree.AddWord("a", { 2 });
    auto result = tree.GetWords(Digits{ 2 });
    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == "a");
}
```

Multiple assertions in one test (example)

- First step in solution: split into two tests

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;
    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);
}

TEST_CASE("Find one letter word which exist in tree", "[Search]")
{
    WordsTree tree;
    tree.AddWord("a", { 2 });
    auto result = tree.GetWords(Digits{ 2 });
    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == "a");
}
```

Here we have two REQUIREs.
Should we use CHECK?
Do we care about the second test if the first test fails?

Multiple assertions in one test (example)

- First step in solution: split into two tests

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;
    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);
}

TEST_CASE("Find one letter word which exist in tree", "[Search]")
{
    WordsTree tree;
    tree.AddWord("a", { 2 });
    auto result = tree.GetWords(Digits{ 2 });
    REQUIRE(result.size() == 1);
    REQUIRE(result[0] == "a");
}
```

In this case, it would probably be better to display the content of the result, even if the first test fails.

Multiple assertions in one test (example)

- Second step in solution: Replace the first REQUIRE with CHECK

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;
    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);
}

TEST_CASE("Find one letter word which exist in tree", "[Search]")
{
    WordsTree tree;
    tree.AddWord("a", { 2 });
    auto result = tree.GetWords(Digits{ 2 });
    CHECK(result.size() == 1);
    REQUIRE(result[0] == "a");
}
```

Multiple assertions in one test (example)

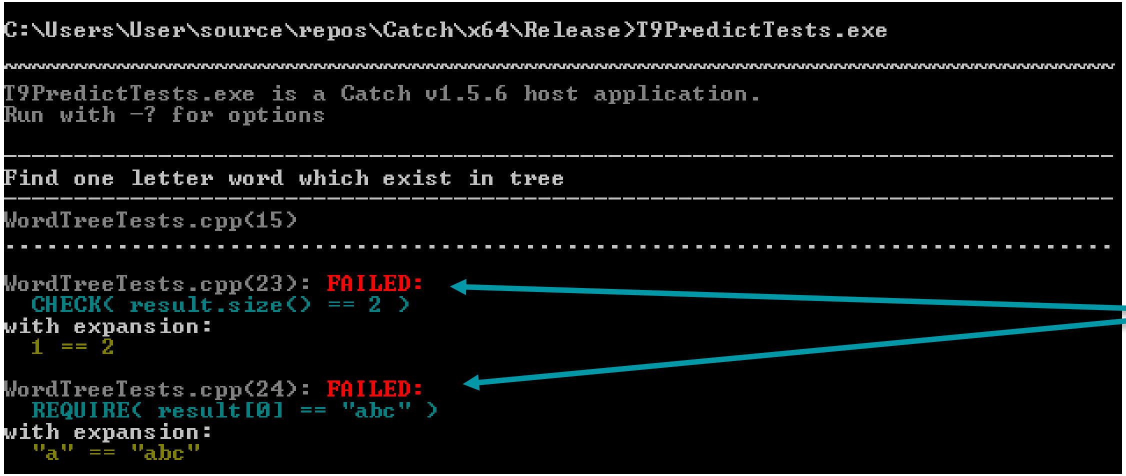
- We changed the last two lines to

```
CHECK(result.size() == 2);
REQUIRE(result[0] == "abc");
```

  so we can see the output of a failed test

Multiple assertions in one test (example)

- Output



```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
T9PredictTests.exe is a Catch v1.5.6 host application.
Run with -? for options

-------------------------------------------------------------------------------
Find one letter word which exist in tree
-------------------------------------------------------------------------------
WordTreeTests.cpp(15)
...............................................................................

WordTreeTests.cpp(23): FAILED:
  CHECK( result.size() == 2 )
with expansion:
  1 == 2

WordTreeTests.cpp(24): FAILED:
  REQUIRE( result[0] == "abc" )
with expansion:
  "a" == "abc"
```

All the outputs are being shown

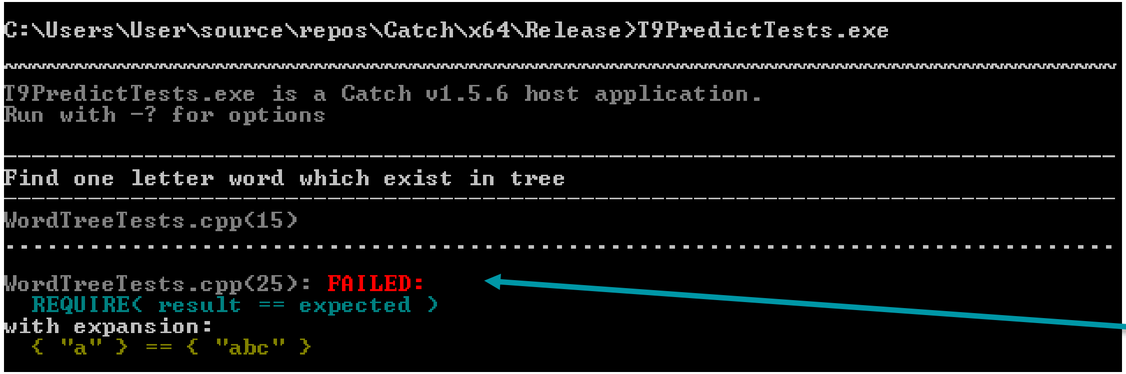
Multiple assertions in one test (example)

- Third step in solution: Refinement of the assert
  by using the `operator==` of the `std::vector` and `std::string`

```
TEST_CASE("If no words exist GetWords returns empty collection", "[Init]")
{
    WordsTree tree;
    REQUIRE(tree.GetWords(Digits{ 1,2,3,4 }).size() == 0);
}

TEST_CASE("Find one letter word which exist in tree", "[Search]")
{
    WordsTree tree;
    tree.AddWord("a", { 2 });
    auto result = tree.GetWords(Digits{ 2 });
    auto expected = vector<string>({ "a" });
    REQUIRE(result == expected);
}
```

Multiple assertions in one test (example)

- Output (after making some changes so that the test fails)



```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
T9PredictTests.exe is a Catch v1.5.6 host application.
Run with -? for options

-------------------------------------------------------------------------------
Find one letter word which exist in tree
-------------------------------------------------------------------------------
WordTreeTests.cpp(15)
...............................................................................

WordTreeTests.cpp(25): FAILED:
  REQUIRE( result == expected )
with expansion:
  { "a" } == { "abc" }
```

The message has all the information that we need

Multiple assertions in one test (example 2)

- Should we split this test into two tests?

```
TEST_CASE("Find word when tree has different encoded word begins with same letter", "[Search]")
{
    WordsTree tree;

    tree.AddWord("ab", { 2, 2 });
    tree.AddWord("ad", { 2, 3 });

    CHECK(tree.GetWords(Digits{ 2, 3 }) == vector<string>({ "ad" }));
    CHECK(tree.GetWords(Digits{ 2, 2 }) == vector<string>({ "ab" }));
}
```

Multiple assertions in one test (example 2)

- The answer is: Yes

```
TEST_CASE("Find word when tree has different encoded word begins with same letter", "[Search]")
{
    WordsTree tree;

    tree.AddWord("ab", { 2, 2 });
    tree.AddWord("ad", { 2, 3 });

    CHECK(tree.GetWords(Digits{ 2, 3 }) == vector<string>({ "ad" }));
    CHECK(tree.GetWords(Digits{ 2, 2 }) == vector<string>({ "ab" }));
}
```

For both of these lines we do two things:

1. Do an action
2. Check the result

Asserting for exceptions

- There are specific macros for exception-checking
  - REQUIRE_THROWS(expression)
  - CHECK_THROWS(expression)
- If the exception is thrown in the expression, the test passes
- If no exceptions are thrown in the expression, the test fails

Asserting for exceptions

- If we want to check the exceptions of a certain type,
  we can use
  - REQUIRE_THROWS_AS(expression, type)
  - CHECK_THROWS_AS(expression, type)
- If the exception of the said type is thrown in the expression,
  the test passes
- If no exceptions are thrown, or
  if the exceptions of the different type is thrown in the expression,
  the test fails

Asserting for exceptions

- If we want to check the exceptions of a certain type,
  we can use
  - REQUIRE_NOTHROW(expression)
  - CHECK_NOTHROW (expression)
- If no exceptions are thrown,
  the test passes
- If the exception of any type is thrown in the expression,
  the test fails
- These are used primarily for specifying
  that the test asserts that the code will not break

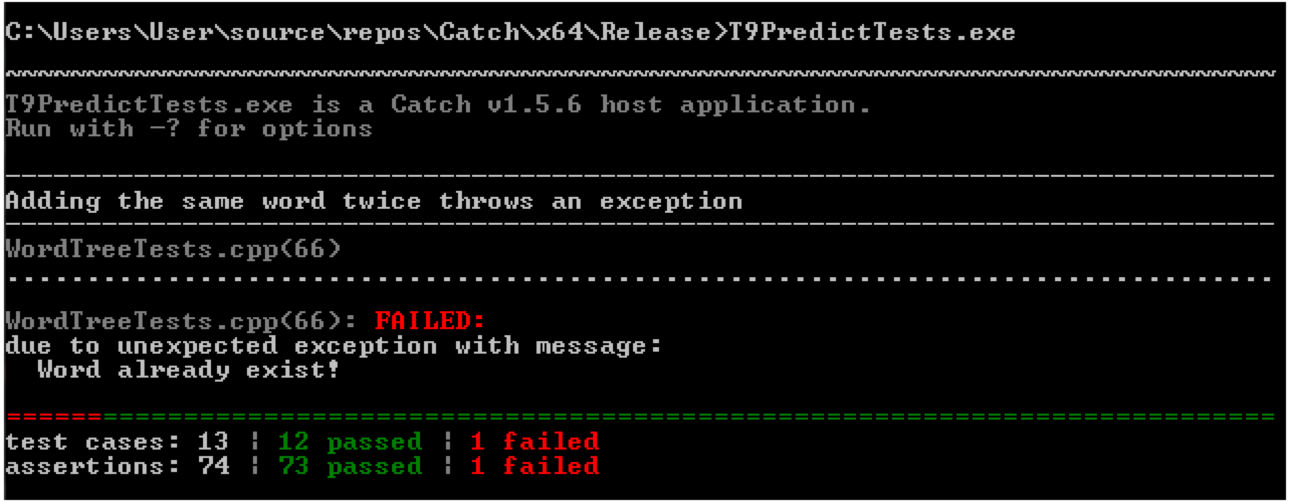
Asserting for exceptions (example)

- The next test throws an exception
  and fails the test
- We want the test to pass,
  if it throws an exception

```
TEST_CASE("Adding the same word twice throws an exception")
{
    WordsTree tree;

    tree.AddWord("ab", { 2, 2 });
    tree.AddWord("ab", { 2, 2 });
}
```

Asserting for exceptions (example)

- Output



```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
T9PredictTests.exe is a Catch v1.5.6 host application.
Run with -? for options

-------------------------------------------------------------------------------
Adding the same word twice throws an exception
-------------------------------------------------------------------------------
WordTreeTests.cpp(66)
...............................................................................

WordTreeTests.cpp(66): FAILED:
due to unexpected exception with message:
  Word already exist!

===============================================================================
test cases: 13 | 12 passed | 1 failed
assertions: 74 | 73 passed | 1 failed
```

Asserting for exceptions (example)

- First step in solution: specify which action should throw an exception

```
TEST_CASE("Adding the same word twice throws an exception")
{
    WordsTree tree;

    tree.AddWord("ab", { 2, 2 });
    REQUIRE_THROWS(tree.AddWord("ab", { 2, 2 }));
}
```

Asserting for exceptions (example)

- Output

```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe
===============================================================================
All tests passed (74 assertions in 13 test cases)
```

Asserting for exceptions (example)

- Second step in solution: specify which exception should be thrown

```
TEST_CASE("Adding the same word twice throws an exception")
{
    WordsTree tree;

    tree.AddWord("ab", { 2, 2 });
    REQUIRE_THROWS_AS(tree.AddWord("ab", { 2, 2 }), WordsTreeException);
}
```

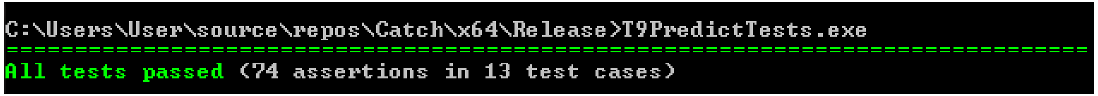
Asserting for exceptions (example)

- Output

```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe
===============================================================================
All tests passed (74 assertions in 13 test cases)
```

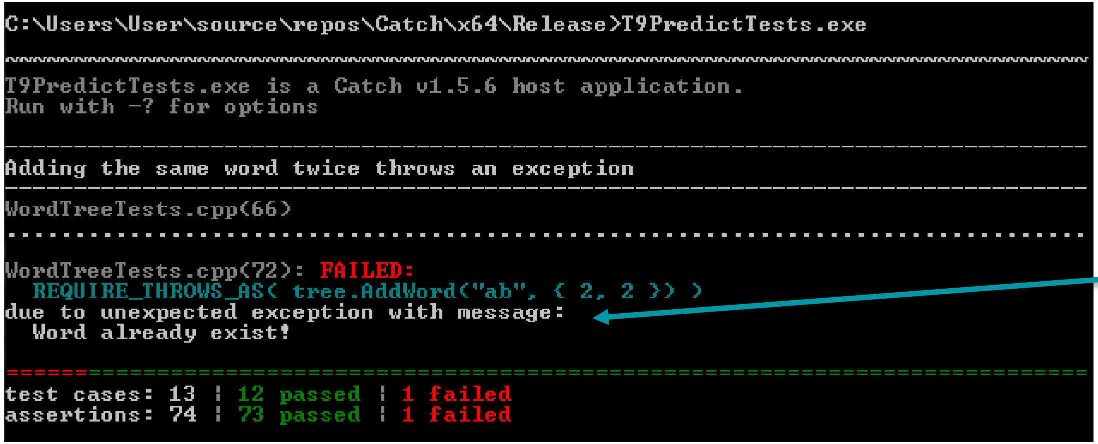
Asserting for exceptions (example)

- Output



```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe
===============================================================================
All tests passed (74 assertions in 13 test cases)
```

- Output (after specifying the wrong type)



```
C:\Users\User\source\repos\Catch\x64\Release>T9PredictTests.exe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
T9PredictTests.exe is a Catch v1.5.6 host application.
Run with -? for options

-------------------------------------------------------------------------------
Adding the same word twice throws an exception
-------------------------------------------------------------------------------
WordTreeTests.cpp(66)
...............................................................................

WordTreeTests.cpp(72): FAILED:
  REQUIRE_THROWS_AS( tree.AddWord("ab", { 2, 2 }) )
due to unexpected exception with message:
  Word already exist!

===============================================================================
test cases: 13 | 12 passed | 1 failed
assertions: 74 | 73 passed | 1 failed
```

It got the exception
it was not supposed to have

Adding more information to test run

- Sometimes we want to supply our information to the test output
- There are four types of messages we can add
  - INFO — the message will only be displayed if the test fails
  - WARN — always shows the message
  - FAIL — always shows the message, and fails the test immediately
  - CAPTURE — used for logging the name and a value of a variable, and works like INFO

Logging macros

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

CAPTURE(someValue); // someValue := 123
```

Simple information from user-defined types

- We can compare the value of a variable with the expected value using the `operator==`
- However, when the test fails,
  we can get test results that looks similar to

```
FAILED:
    REQUIRE( result == expected )
with expansion:
    {?} == {?}
```

String conversions

- There are four ways to change how Catch shows types in assertions and logging expressions:
  - Overloading `operator<<`
  - Adding behavior to `Catch::toString` method
  - Creating a new `Catch::StringMaker` specialization
  - Affecting how the exception are shown using `CATCH_TRANSLATE_EXCEPTION`
- Assumptions:
  - Our class is named `MyType`
  - There is a method/function with declaration `std::string convert(MyType const& value)`

Overloading operator<<

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

Overloading `Catch::toString`

```
namespace Catch
{
    string toString(MyType const& value)
    {
        return convert(value);
    }
}
```

- When we won’t or can’t change the code
- When the class already overloads the `operator<<`

Catch::StringMaker specialization

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

- There are cases when the overloading `Catch::toString` does not work as intended

Custom exception text

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

- `MyType` is the class of the custom exception
- There is a method/function with declaration
  `std::string MyType::message()`