Unit Testing in C++ Using the Catch2 Library

A Presentation by Cliff Green

Oct 17, 2023, updated Oct 18, 2023

(Containing references to Balloon Fiesta, C++ testing, trains, and an occasional gratuitous cat pic)

Cover photo by Cliff Green, Kolibri balloon piloted by John Wahl, Balloon Fiesta 2023

Theme is Framed Pastel Template by PresentationGO, https://www.presentationgo.com

An Overview of the Uplifting Contents

- Catch2 overview and features
- Assertions, the heat engine and beating heart of unit testing
- Example Catch2 test cases
- A digression to note actual bugs found with Catch2 when creating this presentation!
- Wrap up, final thoughts about unit testing
- Gratuitous photos that may pop up here and there

Catch2 Overview

Elvis is aloft! The “special shapes” balloons are entertaining - Star Wars (Darth Vader, Yoda), cartoon characters, many kinds of animals, Smokey Bear, sun and moon, “Pigasus”, a giant sloth, and dozens of others

Photo by Cliff Green, Albuquerque International Balloon Fiesta 2023

Catch2 is a Popular C++ Unit Test Library

- Catch2 is on GitHub: https://github.com/catchorg/Catch2
- Designed to be lightweight with no 3rd party dependencies (although there are smaller / lighter unit test libraries with a subset of functionality)
- An older version of Catch2 is header-only (can make build process easier for some), newest version more traditional `.hpp` and `.cpp` files (for faster compile times)
- Catch2 also has a micro benchmarking facility and BDD (Behavior Driven Development) macros (neither covered in this presentation)

A Simple Example

```
// Slightly modified from Catch2 introduction

#include <catch2/catch_test_macros.hpp>

#include <cstdint>

uint32_t factorial( uint32_t 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) == 3'628'800 );
    REQUIRE( factorial(0) == 1 );
}
```

Output from Previous Example

```
-------------------------------------------------------------------
Factorials are computed
-------------------------------------------------------------------
example.cpp:10
...................................................................

example.cpp:15: FAILED:
  REQUIRE( factorial(0) == 1 )
with expansion:
  0 == 1

===================================================================
test cases: 1 | 1 failed
assertions: 5 | 4 passed | 1 failed
```

Automatically Provided by Catch2

- Descriptive output including testing message and test name and which source file is being tested
- How many assertions passed and how many failed; failed assertions show source line of failure (and actual failure comparison)
- Command line parameters include:
  - Specifying specific tests to run by name
  - Verbose output versus summary output
  - Reporting timings
  - Specifying a different run order for test cases
  - Aborting after a specified number of failures
  - Dozens more

Built-in and Customizable Output Reporters

- General logging messages can be inserted anywhere - `INFO` macro
- Customized output reporters can be written and specified as runtime command line parameter
- Built-in reports include support for multiple formats including JUnit, TeamCity, Automake, TAP, SonarQube, and Bazel
  - Supports many kinds of CI (continuous integration) environments
- Default output returns success to operating system if all assertions succeed, failure otherwise

Other Features

- CMake integration and support
- Floating point comparisons (more in “assertions” slides)
- Throw / catch support and reporting of uncaught exceptions
- Runtime error / crash support (but limited, not as full featured as Boost Test for runtime monitoring)
- “Generators” that can generate simple sets of test data (and a generator framework that can be customized)

Assertions are the Heat Engine of Unit Testing

My son and grandcat Stoney, still a “teenager” cat at the time, getting ready for a Halloween outing as Taco Kitty and Taco Dad

Photo by Nathan Green, Taco Kitty and Taco Dad, Halloween 2021

Require and Check Assertions

- `REQUIRE(expression)` - fail and exit test case if evaluates false
- `CHECK(expression)` - same as `REQUIRE`, but continues execution
  - `REQUIRE_FALSE` and `CHECK_FALSE` simpler than negating expression
  - A thrown exception of any type is a failure
- `REQUIRE_NOTHROW(expression)` (and corresponding `CHECK_NOTHROW`) asserts no exception is thrown
- `REQUIRE_THROWS(expression)` (and corresponding `CHECK_THROWS`) expects an exception (of any type) is thrown
- `REQUIRE_THROWS_AS(expression, exception_type)` expects an exception of the specified type to be thrown (similar for `CHECK_THROWS_AS`)

Floating Point Assertions in Catch2

- As in typical C++, floating point numbers should not be directly compare for equality / inequality with the `REQUIRE` assertions
- `REQUIRE_THAT(value, matcher(target, x))` is used for floating point comparison, where matcher is one of:
  - `WithinAbs(target, margin)`
  - `WithinRel(target, epsilon)`
  - `WithinULP(target, maxUlpDiff)` (max “unit in the last place” difference)
- `IsNaN()` support

More Matcher Assertions in Catch2

- Catch2 has string matcher assertions - `StartsWith`, `EndsWith`, `ContainsSubstring`, `Equals`, and `Matches` (regex match); the match can be case sensitive or insensitive
- Catch2 has `std::vector` matchers for specific elements or complete collections
- Catch2 has a generic predicate matcher, where a callable is provided (e.g. a lambda)
- A set of generic range matchers are available
- Custom matchers can be created

Another Simple Example

```
#include <catch2/catch_test_macros.hpp>

auto square(auto t) { return t * t; } // C++ 20 template syntax


TEST_CASE( "Square generic function, ints", "[square-int][square]" ) {
    REQUIRE( square(0) == 0 );
    REQUIRE( square(1) == 1 );
    REQUIRE( square(2) == 4 );
    REQUIRE( square(-1) == 1 );
    REQUIRE( square(-2) == 4);
    REQUIRE( square(55) == 3); // should fail
}
```

- Note two different test name tags

Output from Previous Example

```
Square generic function, ints
---------------------------------------
example.cpp:7
.......................................

example.cpp:13: FAILED:
  REQUIRE( square(55) == 3 )
with expansion:
  3025 (0xbd1) == 3

=======================================
test cases: 1 | 1 failed
assertions: 6 | 5 passed | 1 failed
```

Catch2 Simplifies C++ Template Test Cases

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/catch_template_test_macros.hpp>
#include <cmath>

auto square(auto t) { return t * t; }

TEST_CASE( "Square, double", "[square-double][square]" ) {
    REQUIRE( square(3.0) == 9.0 );
}

TEMPLATE_TEST_CASE( "Square generic", "[square-generic][square]",
                    int, short, float, double, long double ) {
    REQUIRE( square<TestType>(3) == std::pow(3, 2));
}
```

Simple “String Like” Class, Max Fixed Size

```
// #include <cstddef>, <string>, <string_view>, <array>, <stdexcept>, <algorithm>
template <std::size_t max_sz>
class f_str {
public:
  f_str() = default;
  f_str(std::string_view);
  void append(std::string_view);
  std::string get_str() const noexcept {
    return std::string(m_chars.data(), m_curr_size);
  }
  std::size_t size() const noexcept { return m_curr_size; }
  std::size_t max_size() const noexcept { return max_sz; }
private:
  std::array<char, max_sz> m_chars;
  std::size_t m_curr_size {0};
};
```

`f_str` Constructor and `append` Method Implementation

```
template <std::size_t max_sz>
f_str<max_sz>::f_str(std::string_view s) : m_chars{}, m_curr_size{0} {
  if (s.length() > max_sz) { throw std::range_error("str too big"); }
  std::copy(s.begin(), s.end(), m_chars.begin());
  m_curr_size = s.length();
}

template <std::size_t max_sz>
void f_str<max_sz>::append (std::string_view s) {
  if ((m_curr_size + s.length()) > max_sz) {
    throw std::range_error("appended len too big");
  }
  std::copy(s.begin(), s.end(), (m_chars.begin() + m_curr_size));
  m_curr_size += s.length();
}
```

Photo by Cliff Green, Chehalis Washington, Aug 2007

Minor (But Related!) Digression - Unit Testing Finds Bugs!

I’m a train history buff (and a model railroader). This is a steam train excursion in WA state, we (family) all had a great time on it.

Original Version of `get_str` Method - Do You See the Bug?

```
template <std::size_t max_sz>
class f_str {
// ...
  std::string get_str() const noexcept {
    return std::string(m_chars.begin(), m_chars.end());
  }
// ...
private:
  std::array<char, max_sz> m_chars;
// ...
};
```

- My test case failed - obtaining a `std::string` object from a default constructed `f_str` should have a 0 length

Correct Version of `get_str` Method

```
template <std::size_t max_sz>
class f_str {
// ...
  std::string get_str() const noexcept {
    return std::string(m_chars.data(), m_curr_size);
  }
// ...
private:
  std::array<char, max_sz> m_chars;
// ...
};
```

- Note the first test case (default ctor) in upcoming slides

Original version of `append` Method - Do You See the Bug?

```
template <std::size_t max_sz>
class f_str {
// ...
private:
  std::array<char, max_sz> m_chars;
// ...
};

template <std::size_t max_sz>
void f_str<max_sz>::append (std::string_view s) {
  if ((m_curr_size + s.length()) > max_sz) {
    throw std::range_error("appended len too big");
  }
  std::copy(s.begin(), s.end(), m_chars.end());
  m_curr_size += s.length();
}
```

Correct Version of append Method

```
template <std::size_t max_sz>
class f_str {
// ...
private:
  std::array<char, max_sz> m_chars;
// ...
};
template <std::size_t max_sz>
void f_str<max_sz>::append (std::string_view s) {
  if ((m_curr_size + s.length()) > max_sz) {
    throw std::range_error("appended len too big");
  }
  std::copy(s.begin(), s.end(), (m_chars.begin() + m_curr_size)); // ahh, now correct!
  m_curr_size += s.length();
}
```

Default Ctor Test Case

```
TEST_CASE( "f_str default ctor", "[f_str]" ) {
    f_str<10> f_obj1;
    REQUIRE( f_obj1.size() == 0 );
    REQUIRE( f_obj1.max_size() == 10 );
    auto s1 = f_obj1.get_str();
    REQUIRE( s1.length() == 0 );
    f_str<0> f_obj2;
    REQUIRE( f_obj2.size() == 0 );
    REQUIRE( f_obj2.max_size() == 0 );
    auto s2 = f_obj2.get_str();
    REQUIRE( s2.length() == 0 );
}
```

- Note the duplicate code - this test case can be refactored

Refactored Default Ctor Test Case

```
template <std::size_t sz>
void default_ctor_test() {
    f_str<sz> f_obj;
    REQUIRE( f_obj.size() == 0 );
    REQUIRE( f_obj.max_size() == sz );
    auto s = f_obj.get_str();
    REQUIRE( s.length() == 0 );
}

TEST_CASE( "f_str default ctor", "[f_str]" ) {
    default_ctor_test<0>();
    default_ctor_test<1>();
    default_ctor_test<10>();
    // ... whatever other sizes need to be tested
}
```

Test the `std::string_view` Constructor

```
template <std::size_t sz>
void str_ctor_test(std::string_view s_parm) {
    f_str<sz> f_obj(s_parm);
    REQUIRE( f_obj.size() == s_parm.length() );
    REQUIRE( f_obj.max_size() == sz );
    auto s = f_obj.get_str();
    REQUIRE( s.length() == s_parm.length() );
}

TEST_CASE( "f_str str ctor", "[f_str]" ) {
    str_ctor_test<10>("Howdy!");
    str_ctor_test<7>("Podnah!"); // note - just enough space
    // ... whatever other sizes need to be tested
}
```

Test the append Method

```
template <std::size_t sz>
void append_test(std::string_view s_parm, std::string_view app_parm) {
    f_str<sz> f_obj(s_parm);
    REQUIRE( f_obj.size() == s_parm.length() );
    f_obj.append(app_parm);
    auto s = std::string(s_parm) + std::string(app_parm);
    REQUIRE( f_obj.size() == s.length() );
}
// Note the corner cases being tested, in particular empty strings
TEST_CASE( "f_str append", "[f_str]" ) {
    append_test<13>("Howdy!", "Podnah!"); // note just enough space
    append_test<40>("I enjoyed", " the Balloon Fiesta!");
    append_test<20>("", "Append to empty");
    append_test<30>("Nothing will be appended", "");
}
```

Test the Error Handling - `throw` Logic

```
template <std::size_t sz>
void throw_test_ctr(std::string_view s_parm) {
    REQUIRE_THROWS (f_str<sz>(s_parm));
}
template <std::size_t sz>
void throw_test_append(std::string_view s_parm, std::string_view app_parm) {
    f_str<sz> f_obj(s_parm);
    REQUIRE_THROWS (f_obj.append(app_parm));
}

TEST_CASE( "f_str throw", "[f_str]" ) {
    throw_test_ctr<0>("A");
    throw_test_ctr<5>("Nine char");
    throw_test_append<5>("abcde", "x");
    throw_test_append<0>("", "A");
}
```

Expanding on the Example Code and Assertions

- The `f_str` class template example is incomplete - it should have additional functionality to ensure it is a regular type
- The assertions in the `f_str` examples primarily test for length correctness - object content should also have assertions
- The examples show both production and test code next to each other - typically the test code will include appropriate production headers and build system will link everything together (i.e. production and test code are separate and in independent folders)

Final Thoughts on Unit Testing

A beautiful pic of a classic Santa Fe FT unit, being washed in Argentine, Kansas, 1943. I had a Lionel toy train version of this loco when I was a wee little kid.

Photo by Jack Delano, US Library of Congress, https://commons.wikimedia.org/w/index.php?curid=3598747

Unit Testing Has Many Pros

- I found and fixed two subtle (and significant) bugs while creating this presentation
- Once written, test code is executed again and again, and with CI (continuous integration) test code is run any time changes happen in a repository
- Designing code from the perspective of the user of the code clarifies and improves the interfaces
- Design by contract is reinforced with unit test code - note the invariants and preconditions that were tested in the `f_str` examples

Consider Other Ways of Testing Code

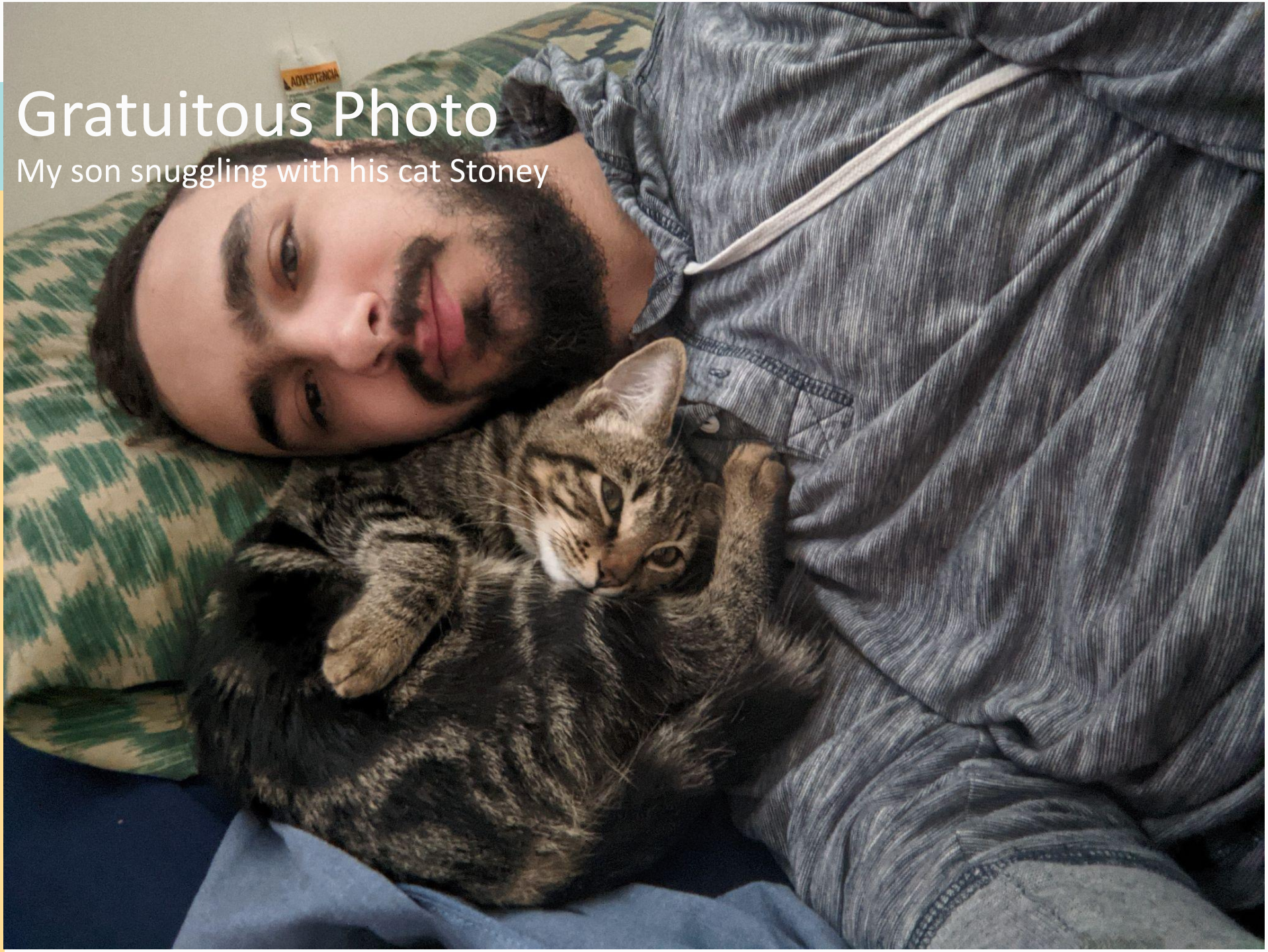
- Log / print messages scattered throughout code (clutters runtime environment!)
- Running code in a debugger (slow and laborious!)
- Manually running the executable and “watching what happens” (also laborious!)
- All of the above may still be needed at times (although I hardly ever use a debugger), but are slow and repetitive and rarely cover more than a few basic data cases or edge conditions

Unit Testing Has Some Cons

- It takes time to design and write unit tests; production code takes longer until it is ready (but tends to have less bugs)
- Unit tests are only as good as the unit test design and code - sometimes the test code is buggy, not the production code
- Unit testing often times cannot cover all functionality, depending on the system and overall complexity
- Build scripts / CMake files have to be created for the unit tests, not just the production code

Gratuitous Photo

My son snuggling with his cat Stoney



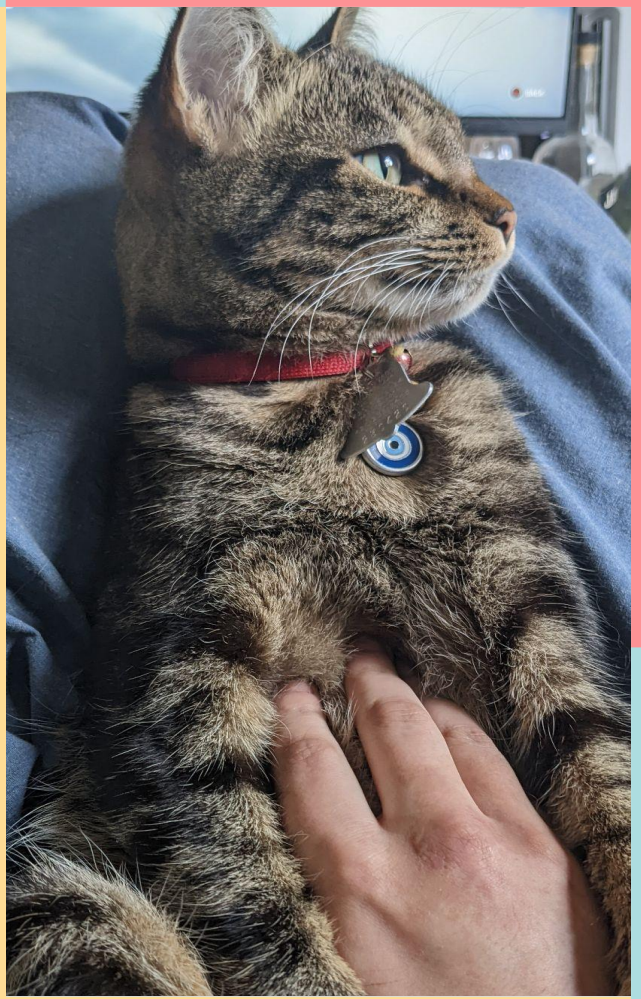


Photo by Nathan Green

Couldn’t Resist One More Gratuitous Cat Photo

Grandcat Stoney getting a tummy rub

Thank You!

- Two Essential Online Tools (mentioned in all of my C++ presentations):
  - CPP Reference page: https://en.cppreference.com/w/- have it ready, learn to read / use it
  - Compiler Explorer: https://godbolt.org/ - compile and run your code on multiple compilers, analyze the assembler output
- Question and discussion time