Reasons for Using Guidelines

- Anti-bikeshed
- You can hurt yourself on this; do it this way to be safe
- Stop using that feature, use this one instead
- Here's how that new thing works
- We wrote a handy library; use it please

Stop the Bikeshedding

C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead.

C.48: Prefer in-class initializers to member initializers in constructors for constant initializers

```
class Simple
{
public:
    Simple() : a(1), b(2), c(3) {}
    Simple(int aa, int bb, int cc=-1) : a(aa), b(bb), c(cc)
{}
    Simple(int aa) { a = aa; b = 0; c=0; }
private:
    int a;
    int b;
    int c;
};
```

```
class Simple
{
public:
    Simple()  {}
    Simple(int aa, int bb, int cc) : a(aa), b(bb), c(cc)
{}
    Simple(int aa) : a(aa) {}
private:
    int a = -1;
    int b = -1;
    int c = -1;
};
```

```
class Simple
{
public:
    Simple() = default;
    Simple(int aa, int bb, int cc) : a(aa), b(bb), c(cc)
{}
    Simple(int aa) : a(aa) {}
private:
    int a = -1;
    int b = -1;
    int c = -1;
};
```

Benefits

- No arguing about “equivalent” ways to do it
- May prevent some bugs
- May put you back in “compiler generates constructors” land
- Potentially marginally faster in some circumstances

F.51: Where there is a choice, prefer default arguments over overloading.

```
class Reactor
{
public:
        double Offset(double a, double b, double ff);
        double Offset(double a, double b);
// .. rest of the class
};
```

```
double Reactor::Offset(double a, double b, double ff)
{
    //insanely complicated calculations using member variables, a, b, and ff
    return whatever;
}
double Reactor::Offset(double a, double b)
{
    return Offset(a,b,1.0);
}
```

```
class Reactor
{
public:
         double Offset(double a, double b, double ff=1.0);
// .. rest of the class
};
```

```
double Reactor::Offset(double a, double b, double ff /* = 1.0*/)
{
    //insanely complicated calculations using member variables, a, b, and ff
    return whatever;
}
```

Benefits

- No arguing about “equivalent” ways to do it
- Will not forget to make same change to both copies
- Difference between the two “versions” is crystal clear

Do Not Run With Scissors

C.47: Define and initialize member variables in the order of member declaration.

```
class Wrinkle
{
public:
    Wrinkle(int i) : a(++i), b(++i), x(++i) {}
private:
    int a;
    int x;
    int b;
};

int main()
{
    Wrinkle w(0);
    return 0;
}
```

Sure, that’s fake, but…

- `FullName` initialized from `FirstName` and `LastName`
- … `p(new Foo), x(p->Something())`

Who rearranges declarations?

- Helpful tools
- Eager new employees
- Putting variables in alphabetical order
- Grouping like with like

Benefits

- Protects you from an oddity of the language without requiring everyone to know it
- Might encourage you to rethink your class design so this dependency goes away

I.23: Keep the number of function arguments low

```
int area(int x1, int y1, int x2, int y2);
int a = area(1, 1, 11, 21);

int area(Point p1, Point p2);
int a = area({ 1,1 }, { 11,21 });

class Customer
{
   Person details;
   Salesrep rep;
};


Customer(Person p, Salesrep s);

Customer(string pfirst, string plast, string pph,
string sfirst, string slast, string sph, string sid);
```

Benefits

- Lower cognitive burden and recall effort
- Introduce abstractions that may be useful later
- Perhaps save ripple changes


That Old Thing?
Nobody Goes There Anymore

ES.50: Don't cast away const.

```
class Stuff
{
private:
    int number1;
    double number2;

    int LongComplicatedCalculation() const;

public:
    Stuff(int n1, double n2) : number1(n1), number2(n2)
{}
    bool Service1(int x);
    bool Service2(int y);
    int getValue() const;

};
```

```
class Stuff
{
private:
    int number1;
    double number2;
    int LongComplicatedCalculation() const;
    int cachedValue;

public:
    Stuff(int n1, double n2) : number1(n1), number2(n2),
cachedValue(0) {}
    bool Service1(int x);
    bool Service2(int y);
    int getValue() const;
};
```

```
bool Stuff::Service1(int x)
{
    number1 = x;
    //other real calculations
    cachedValue = LongComplicatedCalculation();
    return true;
}
bool Stuff::Service2(int y)
{
    number2 = y / 3.0;
    //other real calculations
    cachedValue = LongComplicatedCalculation();
    return true;
}
int Stuff::getValue() const
{
    return cachedValue;
}
```

```
class Stuff
{
private:
    int number1;
    double number2;

    int LongComplicatedCalculation() const;
    int cachedValue;
    bool cacheValid;

public:
    Stuff(int n1, double n2) : number1(n1), number2(n2),
cachedValue(0), cacheValid(false) {}
    bool Service1(int x);
    bool Service2(int y);
    int getValue() const;

};
```

```
bool Stuff::Service1(int x)
{
    number1 = x;
    //other real calculations
    cacheValid = false;
    return true;
}
bool Stuff::Service2(int y)
{
    number2 = y / 3.0;
    //other real calculations
    cacheValid = false;
    return true;
}
int Stuff::getValue() const
{
    if (!cacheValid)
    {
        cachedValue = LongComplicatedCalculation();
        cacheValid = true;
    }
    return cachedValue;
}
```

Restoring const-correctness

- Take const off of getValue()
  - Might break code that calls it on const instances
  - Giving away something that was useful in the design
- Cast away const in getValue()
  - Allows getValue to change **anything** in the instance
  - Header file is now a lie

Restoring const-correctness

- Replace pair of cache-related variables with a pointer to a cache instance. The pointer is const, what it points to is not, change the isValid and Value of it with abandon
  - Ok, but introduces complexity
  - Good if the abstraction (eg cache) is something you can name
- Mark these variables as mutable
  - Header tells the truth
  - getValue can only change the mutable variables

```
class Stuff
{
private:
    int number1;
    double number2;
    int LongComplicatedCalculation() const;
    mutable int cachedValue;
    mutable bool cacheValid;
 public:
    Stuff(int n1, double n2) : number1(n1),
number2(n2),cachedValue(0),cacheValid(false) {}
    bool Service1(int x);
    bool Service2(int y);
    int getValue() const;
};
```

Benefits

- Code accessing mutable members is shorter and more readable
- Easier to write, read, and maintain
- Const-correctness may enable optimizations

I.11: Never transfer ownership by a raw pointer (T*)

Not like this…

```
Policy* SetupAndPrice(args)
{
    Policy * p = new Policy{...};
    // ...
    return p;
}
```

Alternatives to raw pointers

- Return by value
  - You don’t mind a copy (which may be elided)
- Take by non-const reference and change it
  - Best if you’re const-correct throughout so absence speaks volumes
  - Prefer this when the parameter had a purpose before the call

Alternatives to raw pointers

- Return an appropriate smart pointer
  - Will manage ownership and lifetime for you
- Use owner<> from GSL
  - Tells analysis tools who is responsible for cleanup
  - Also tells humans who use the API
  - Great for calls you can’t change

gsl::owner<>

```
template <class T, class =
std::enable_if_t<std::is_pointer<T>::value>>

using owner = T;
```

Benefits

- Memory management is too important to do entirely in your head
  - Don’t do it at all
  - Or get code to do it
  - Or at least get a marker to remind you to do it

All the Cool Kids are Using It

F.21: To return multiple "out" values, prefer returning a tuple or struct.

```
int foo(int inValue, int& outValue)
{
    outValue = inValue * 2;
    return inValue * 3;
}

int main()
{
    int number = 4;
    int answer = foo(5, number);
    return 0;
}
```

Your own struct

```
struct twoNumbers
{
    int value1;
    int value2;
};

twoNumbers fooStruct(int inValue)
{
    return twoNumbers{ inValue * 2, inValue * 3 };
}

int main()
{
    int number, answer;
    twoNumbers result = fooStruct(6);
    number = result.value1;
    answer = result.value2;
    return 0;
}
```

std::optional

- If the two things are [an object] and [a bool about whether or not that object is usable] consider `optional<T>`
- Casting an `optional` to `bool` returns true if it contains a value, false otherwise. Other uses (`=`, `->`, `*`, and `value()`) get the value.
- Handy `value_or()` function too

tuple, tie, structured bindings

```
std::tuple<int, int> fooTwo(int inValue)
{
    return std::make_tuple(inValue * 2, inValue * 3);
}

int main()
{
    int number, answer;
    std::tie(answer, number) = fooTwo(9);
    return 0;
}
```

```
int main()
{
    auto[answer, number] = fooTwo(9);
    return 0;
}
```

Benefits

- Readability: all the returns are together
  - Not mixed between one special return value and the rest as out params
- When parameters are passed by pointer or non-const ref, readers can assume they are in-out parameters

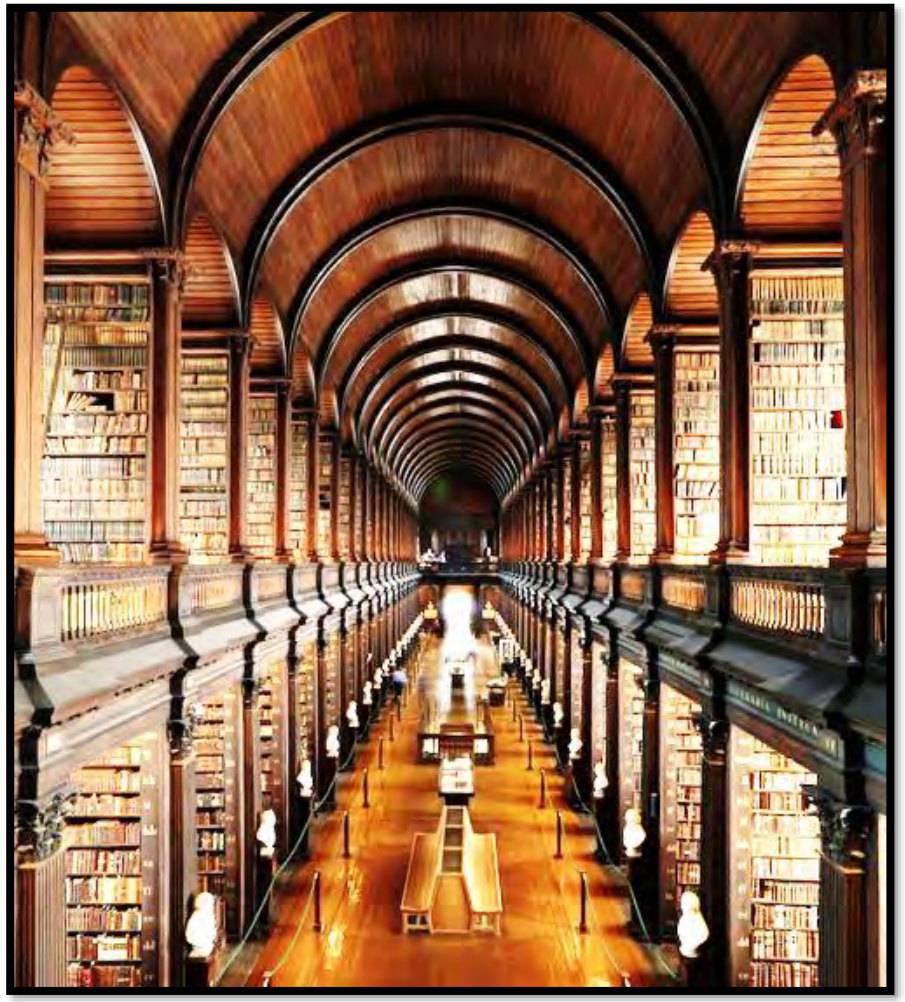
Enum.3: Prefer class enums over "plain" enums

enum class

```
enum class Error {OK, FileNotFound, OutOfMemory};
enum class Ratings{Terrible, OK, Terrific};

enum oldStyle {OH, OK, OR};


Error result = Error::OK;
Ratings stars = Ratings::OK;
int r = static_cast<int>(result);

oldStyle Oklahoma = OK;
oldStyle Ohio = oldStyle::OH;

int x = Ohio;
```

Benefits

- Names can overlap
  - You can have an OK in every enum if you want!
- “Helpful” conversions to int do not happen
  - Use static_cast<> if you want to convert
  - Compiler is your friend when passing to functions
- Can use a backing type other than int if you want



You Are Not The First Developer With This Problem

I.12: Declare a pointer that must not be null as not_null.

```
Service s(1);
Service* ps = &s;
i = ps->DoSomething();
ps = nullptr;
i = ps->DoSomething();
```

```
int AskServiceToDoSomething(Service* p)
{
    if (p)
    {
        return p->DoSomething();
    }
    return -1; //or other signal value or default
}
```

```
#include "gsl/gsl"
// . . .

gsl::not_null<Service*> ps = &s;

i = ps->DoSomething();
ps = nullptr;
```

```
#include "gsl\gsl.h"
// . . .

Service* GetPointer(Service* p)
{
    return p;
}
// . . .

    gsl::not_null<Service*> ps = &s;

    ps = GetPointer(nullptr);

// . . . Thousands of lines later

    i = ps->DoSomething();
```

Benefits

- Avoid dereferencing `nullptr` errors
- Improve performance by avoiding redundant checks for `nullptr`
- Express intent to readers

ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions

…The guideline support library offers a `narrow_cast` for specifying that narrowing is acceptable and a `narrow` that throws an exception if it would throw away information.

```
SYSTEMTIME st;

st.wMilliseconds = 0;
#pragma warning(push)
#pragma warning(disable: 4244) // possible loss of data
- but seconds, etc, can't be too large for a WORD and
will be positive
st.wSecond = inputTime.tm_sec;
st.wMinute = inputTime.tm_min;
st.wHour = inputTime.tm_hour;
st.wDay = inputTime.tm_mday;
st.wMonth = inputTime.tm_mon + 1;
st.wYear = inputTime.tm_year + 1900;
#pragma warning(pop)
```

narrow_cast and narrow

- Use like `static_cast<>`
- Name alone carries information
- `narrow_cast<>` means I don't mind losing data
- `narrow<>` throws error on losing data (the cast changed the value)

Benefits

- Express intent to readers
- Permits checkers to distinguish between kinds of casts
- Can get runtime errors if reasoning about cast being ok was wrong

Summary

- Bikeshedding is bad
  - C.45: in-class member initializers
  - F.51: default arguments
- Don’t hurt yourself
  - C.47: initialize member variables
  - I.23: Keep the number of function arguments low
- Stop using that
  - ES.50: Don't cast away const.
  - I.11: ownership by raw pointer
- Use this new thing properly
  - F.21: return a tuple
  - Enum.3: class enums
- Guideline Support Library
  - I.12: not_null
  - ES.46: narrow_cast and narrow

Call to Action

- Bookmark https://github.com/isocpp/CppCoreGuidelines and https://github.com/Microsoft/GSL
- Search the guidelines next time you have a decision to make
  - Make your decision
  - Use something from the GSL?
- After two or three “good hits” try reading the whole thing
- Consider using a checker to let you know if you’re breaking some

2017 CPP-Summit

Mix Tests and Production Code With Doctest - Implementing and Using the Fastest Modern C++ Testing Framework

Viktor Kirilov

Senior C++ developer

About me

- my name is Viktor Kirilov - from Bulgaria
- 4 years of professional C++ in the games / VFX industries
- working on personal projects since 01.01.2016 (last 2 years)
- some consulting and contract work

Tools of the trade

- compilers: Visual Studio, GCC, Clang, Emscripten
- tools: CMake, Python, git, clang-format, valgrind, sanitizers
- services: GitHub, Travis CI, AppVeyor

Passionate about

- game development and game engines
- data-oriented design and HPC
- good software development practices

What is doctest

The lightest feature-rich C++ single-header testing framework

Inspired by the ability of compiled languages such as **D** / **Rust** / **Nim** to write tests directly in the production code

Project mantra:

Tests can be considered a form of documentation and should be able to reside near the code which they test

Nothing is better than documentation with examples.

Nothing is worse than outdated examples that don't actually work.

Some info

catchorg / Catch2 | Watch 356 | Star 5,963 | Fork 868

onqtam / doctest | Watch 54 | Star 996 | Fork 68

Interface and functionality modeled mainly after Catch and Boost.Test / Google Test

Currently some big things which Catch has are missing:

- reporter system - to file, to xml, user defined
- matchers

but doctest is catching up - and is adding some of its own - like test suites and templated test cases

https://github.com/martinmoene/catch-lest-other-comparison

This presentation

- Introduction to the framework
- Implementation details - cool C++ stuff
  - Automatic test registration
  - The preprocessor and silencing warnings
  - Removing testing-related stuff from the binary
  - Expression decomposition with templates
  - Exception translation
  - Other notable things
- Compile time and runtime benchmarks
- Examples of integration with production code
- **Not** covered: How to do testing - there are plenty of talks on that topic

Single header with 2 parts

```
#ifndef GUARD_FWD
#define GUARD_FWD
// fwd stuff...
#endif // GUARD_FWD


#if defined(DOCTEST_CONFIG_IMPLEMENT)
#ifndef GUARD_IMPL
#define GUARD_IMPL

#include <vector>
// test runner stuff...

#endif // GUARD_IMPL
#endif // DOCTEST_CONFIG_IMPLEMENT
```

- no inline functions and leaked dependencies / headers
- no skyrocketing compile / link times

A complete example


```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>

int fact(int n) { return n <= 1 ? n : fact(n - 1) * n; }

TEST_CASE("testing the factorial function") {
    CHECK(fact(0) == 1); // will fail
    CHECK(fact(1) == 1);
    CHECK(fact(2) == 2);
    CHECK(fact(10) == 3628800);
}
```

Example output

```
[doctest] doctest version is "1.2.6"
[doctest] run with "--help" for options
===============================================================
main.cpp(6)
testing the factorial function

main.cpp(7) FAILED!
  CHECK( fact(0) == 1 )
with expansion:
  CHECK( 0 == 1 )

===============================================================
[doctest] test cases:    1 |    0 passed |    1 failed |
[doctest] assertions:    4 |    3 passed |    1 failed |
```

What makes doctest different

In 2 words: **light** and **unintrusive** (**transparent**):

- The smallest possible footprint on compile times
- Can remove everything testing-related from the binary
- No namespace pollution
- All macros are (or can be) prefixed
- Doesn't drag any headers with it
- 0 warnings
- Easy integration with user code

Unnoticeable even if included in every source file of a project

Very reliable - per commit tested

All tests are built in Debug / Release and in 32 / 64 bit modes.

- GCC: 4.4 / 4.5 / 4.6 / 4.7 / 4.8 / 4.9 / 5 / 6 / 7 (Linux / OSX)
- Clang: 3.5 / 3.6 / 3.7 / 3.8 / 3.9 / 4 / 5 (Linux / OSX)
- MSVC: 2008 / 2010 / 2012 / 2013 / 2015 / 2017
- warnings as errors - on the most aggressive levels
- output compared to one from a previous known good run
- ran through **valgrind** (Linux only)
- ran through address and UB **sanitizers** (Linux / OSX)
- C++98 and C++11, -fno-exceptions, -fno-rtti
- analyzed with 5 different static analyzers

A total of 330+ different configurations are built and tested.

Using travis and appveyor for CI - integrated with GitHub.

Tests in the production code is feasible!

This leads to:

- lower barrier for writing tests (no separate **.cpp** files)
- tests can be viewed as up-to-date comments
- easier testing unexposed internals through public API / headers
- TDD in C++ has never been easier!

The framework can still be used like any other even if the idea of writing tests in the production code doesn't appeal to you.

Other most notable features

One core assertion macro

```
CHECK(a == 666);
CHECK(b != 42);
```

vs

```
CHECK_EQUAL(a, 666);
CHECK_NOT_EQUAL(b, 42);
```

Other most notable features

Automatic test case registration

```
TEST_CASE("name") {
    // asserts
}
```

vs

```
TEST_CASE(unique_identifier, "name") {
    // asserts
}

void some_function_called_from_main() {
    doctest::register(unique_identifier);
}
```

Other most notable features

Subcases for shared setup/teardown

```
TEST_CASE("db") {
    auto db = open("...");

    SUBCASE("first tests") {
        // asserts 1 with db
    }

    SUBCASE("second tests") {
        // asserts 2 with db
    }

    close(db);
}
```

VS

```
TEST_CASE("db - first tests") {
    auto db = open("...");

    // asserts 1 with db

    close(db);
}

TEST_CASE("db - second tests") {
    auto db = open("...");

    // asserts 2 with db

    close(db);
}
```

Other most notable features

logging facilities with lazy stringification for performance

```
for(int i = 0; i < 100; ++i) {
    INFO("the value of i is " << i);
    CHECK(a[i] == b[i]);
}
```

will output the following:

```
test.cpp(10) ERROR!
  CHECK( a[i] == b[i] )
with expansion:
  CHECK( 0 == 32762 )
with context:
  the value of i is 75
```

Other most notable features

translation of exceptions

```
int func() { throw MyType(); return 0; }

REG_TRANSLATOR(const MyType& e) {
    return String("MyType: ") + toString(e);
}

TEST_CASE("foo") {
    CHECK(func() == 42);
}
```

will output the following:

```
main.cpp(34) ERROR!
  CHECK( func() == 42 )
threw exception:
  MyType: contents...
```

Other most notable features

stringification of user types

```
struct type { bool data; };
bool operator==(const type& lhs, const type& rhs) {
    return lhs.data == rhs.data;
}
doctest::String toString(const type& in) {
    return in.data ? "true" : "false";
}

TEST_CASE("stringification") {
    CHECK(type{true} == type{false});
}
```

will output the following:

```
test.cpp(15) ERROR!
  CHECK( type{true} == type{false} )
with expansion:
  CHECK( true == false )
```

Other most notable features

templated test cases

```
typedef doctest::Types<int, char, myType> types;

TEST_CASE_TEMPLATE("serialization", T, types) {
    auto var = T{};
    json state = serialize(var);
    T result = deserialize(state);
    CHECK(var == result);
}
```

will result in the creation of 3 test cases:

- serialization<int>
- serialization<char>
- serialization<myType>

Other most notable features

asserts for exceptions and floating point

```
void throws() { throw 5; }

TEST_CASE("stringification")
{
    CHECK_THROWS(throws());
    CHECK_THROWS_AS(throws(), int);
    CHECK_NOTHROW(throws());

    CHECK(doctest::Approx(5.f) == 5.001f);
}
```

Other most notable features

decorators for test cases and test suites

```
bool is_slow() { return true; }

TEST_CASE("should be below 200ms"
    * doctest::skip(is_slow())
    * doctest::timeout(0.2))
{}
```

- may_fail(bool)
- should_fail(bool)
- expected_failures(int)
- description(const char*)
- test_suite(const char*)

Other most notable features

- crash handling with signals (Unix) / SEH (Windows)
- failures can break into the debugger
- command line with lots of options - filtering, colors, etc.
  - tests.exe --list-test-cases // list test case names
  - tests.exe --test-case=*math*,util_* // execute only matching
  - tests.exe --test-suite-exclude=*deprecated* // skip these
  - tests.exe --abort-after=10 // stop tests after 10 failures
  - tests.exe --order-by=rand // can also give seed
  - tests.exe --no-breaks // don't break in debugger
- range-based execution of tests - for parallelization
  - tests.exe --count // get the number of tests
  - tests.exe --first=0 --last=9 // execute first range
  - tests.exe --first=10 --last=19 // execute second range

Let's get into details

Code is simplified for readability

Unique anonymous variables

```
#define CONCAT_IMPL(s1, s2) s1##s2
#define CONCAT(s1, s2) CONCAT_IMPL(s1, s2)

#define ANONYMOUS(x) CONCAT(x, __COUNTER__)

int ANONYMOUS(ANON_VAR_); // int ANON_VAR_5;
int ANONYMOUS(ANON_VAR_); // int ANON_VAR_6;
```

__COUNTER__ yields a bigger integer each time it gets used

non-standard but present in all modern compilers

Auto registration

```
TEST_CASE("math") {
    // asserts
}
```

gets expanded to

```
static void ANON_FUNC_24();          // fwd decl
static int ANON_VAR_25 = regTest( // register
    ANON_FUNC_24, "main.cpp", 56, "math", ts::get());

void ANON_FUNC_24() {                // the test case
    // asserts
}
```

static to not clash during linking with other symbols

Auto registration

```
std::set<TestCase>& getTestRegistry() {
    static std::set<TestCase> data; // static local
    return data; // return a reference
}

int regTest(void (*f)(void) f, const char* file, int line
            const char* name, const char* test_suite)
{
    TestCase tc(name, f, file, line, test_suite);
    getTestRegistry().insert(tc);
    return 0; // to initialize the dummy int
}
```

The test registry of the test runner resides in a special getter to work around the static initialization order fiasco.

Test Suites

```
namespace ts { inline const char* get() { return ""; } } // default

TEST_SUITE("math") {
    TEST_CASE("addition") { // calls ts::get()
        // ...
    }
}
```

after the preprocessor:

```
namespace ts { inline const char* get() { return ""; } } // default

namespace ANON_TS_45 {
    namespace ts { static const char* get() { return "math"; } }
}
namespace ANON_TS_45 {
    TEST_CASE("addition") { // calls ts::get() ==> ANON_TS_45::ts::get()
        // ...
    }
}
```

Lets talk about warnings

The framework and it's tests are clean from these:

- **-Weverything** for Clang
- **/Wall** for MSVC - except for a few:
  - C4514 - removed unreferenced inline function
  - C4571 - SEH related
  - C4710 - function not inlined
  - C4711 - function selected for automatic inline expansion
- **-Wall -Wextra -pedantic** for GCC - and over **37** other unique flags not covered by these!

The additional GCC flags

-Wswitch-default	-Wmissing-include-dirs	-Wnoexcept
-Wconversion	-Wcast-align	-Wtrampolines
-Wold-style-cast	-Wswitch-enum	-Wzero-as-null-pointer-constant
-Wfloat-equal	-Wnon-virtual-dtor	-Wuseless-cast
-Wlogical-op	-Wctor-dtor-privacy	-Wshift-overflow=2
-Wundef	-Wsign-conversion	-Wnull-dereference
-Wredundant-decls	-Wdisabled-optimization	-Wduplicated-cond
-Wshadow	-Weffc++	-Wduplicated-branches
-Wstrict-overflow=5	-Wdouble-promotion	-Wformat=2
-Wwrite-strings	-Winvalid-pch	-Walloc-zero
-Wpointer-arith	-Wmissing-declarations	-Walloca
-Wcast-qual	-Woverloaded-virtual	-Wrestrict

To get the list of enabled / disabled warnings - as seen in

http://stackoverflow.com/questions/11714827/#34971392

g++ -Wall -Wextra -Q --help=warning

Silencing warnings in the header

```
#if defined(__clang__)
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wpadded"
#endif // __clang__

// ... header stuff

#if defined(__clang__)
#pragma clang diagnostic pop
#endif // __clang__
```

Every (decent) compiler can do this.

Warnings in user code?

The **TEST_CASE** macro produces warnings because of the anonymous dummy int:

- clang: **-Wglobal-constructors**
- gcc: **-Wunused-variable**

We cannot ask the user to use pragmas to silence warnings....

What to do?

The preprocessor

```cpp
// test.cpp
#include "test.h"
int main() {}

#pragma pack(2)
struct T { char c; short s; };
```

```cpp
// test.h
#define val(x) x
int a = val(5); // comment
int a = 10; // will get error
```

And after the preprocessor:

```cpp
# 1 "test.h" 1

int a = 5;
int a = 10;
# 3 "test.cpp" 2
int main() {}

#pragma pack(2)
struct T { char c; short s; };
```

Embedding a pragma in a macro

```
// test.cpp
#include <cmath>

#define myParallelTransform(op)   \
    _Pragma("omp parallel for")   \
    for(int n = 0; n < size; ++n) \
      data[n] = op(data[n])

int main() {
    float data[] = {0, 1, 2, 3, 4, 5}
    int size = 6;

    myParallelTransform(sin);
    myParallelTransform(cos);
}
```

```
...

int main() {
    float data[] = {0, 1, 2, 3, 4, 5};
    int size = 6;

#pragma omp parallel for
    for(int n = 0; n < size; ++n)
        data[n] = sin(data[n]);

#pragma omp parallel for
    for(int n = 0; n < size; ++n)
        data[n] = cos(data[n]);
}
```

_Pragma() was standardized in C++11 but compilers support it for many years (**__pragma()** for MSVC)

Silencing warnings in macros

```
#define TEST_CASE_IMPL(f, name)                                          \
    static void f();                                                     \
                                                                         \
    _Pragma("clang diagnostic push")                                     \
    _Pragma("clang diagnostic ignored \"-Wglobal-constructors\"")        \
                                                                         \
    static int ANONYMOUS(ANON_VAR_) =                                    \
            regTest(f, __FILE__, __LINE__, name, ts::get());             \
                                                                         \
    _Pragma("clang diagnostic pop")                                      \
                                                                         \
    void f()

#define TEST_CASE(name) TEST_CASE_IMPL(ANONYMOUS(ANON_FUNC_), name)
```

Macro indirection needed so the same anon name is used.

Silencing warnings in macros

```
#define TEST_CASE_IMPL(f, name)                                          \
    static void f();                                                     \
                                                                         \
    static int ANONYMOUS(ANON_VAR_) __attribute__((unused)) =            \
            regTest(f, __FILE__, __LINE__, name, ts::get());             \
                                                                         \
    void f()

#define TEST_CASE(name) TEST_CASE_IMPL(ANONYMOUS(ANON_FUNC_), name)
```

_Pragma() in the C++ frontend of GCC (g++) isn't working in macros for quite some time (6+ years) - or does only sometimes

- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=55578
- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=69543
- https://github.com/catchorg/Catch2/issues/870

Subcases - a DFS traversal

```
TEST_CASE("nested subcases") {
    out("setup");

    SUBCASE("") {
        out("1");

        SUBCASE("") {
            out("1.1"); // leaf
        }
    }
    SUBCASE("") {
        out("2");

        SUBCASE("") {
            out("2.1"); // leaf
        }
        SUBCASE("") {
            out("2.2"); // leaf
        }
    }
}
```

```
// THE OUTPUT



setup
1
1.1



setup
2
2.1

setup
2
2.2
```

Subcase macro expansion

```
SUBCASE("foo") {
    // ...
    SUBCASE("bar") {
        // ...
    }
    SUBCASE("baz") {
        // ...
    }
}
```

```
if(const Subcase& ANON_2 = Subcase("foo", "a.cpp", 4)) {
    // ...
    if(const Subcase& ANON_3 = Subcase("bar", "a.cpp", 6)) {
        // ...
    }
    if(const Subcase& ANON_4 = Subcase("baz", "a.cpp", 9)) {
        // ...
    }
}
```

- The lifetime of each Subcase is only in the "**then**" blocks.
- The magic happens in the **Ctor** / **Dtor** of the ***Subcase*** class.
- **operator bool()** is used to decide whether to enter the "**if**".
- Subcases are lazily discovered - unlike test cases.
- The DFS traversal is done using globals (hash tables, etc.)
- Can be nested infinitely - and the entered ones are in a stack

The main() entry point


```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>
```

vs

```
#define DOCTEST_CONFIG_IMPLEMENT
#include <doctest.h>
int main(int argc, char** argv) {
    doctest::Context context;
    // default
    context.setOption("abort-after", 5);  // stop after 5 failed asserts
    // apply argc / argv
    context.applyCommandLine(argc, argv);
    // override
    context.setOption("no-breaks", true); // don't break in the debugger
    // run queries or test cases unless with --no-run
    int res = context.run();
    if(context.shouldExit()) // query flags (and --exit) rely on this
        return res;          // propagate the result of the tests
    // your program
    return res; // + your_program_res
}
```

Removing everything testing-related

```
#define DOCTEST_CONFIG_DISABLE // the magic identifier
#include <doctest.h>
```

This results in:

```
#define TEST_CASE(name)                                  \
    template <typename T>                                \
    static inline void ANONYMOUS(ANON_FUNC_)()
```

So all test cases are turned into uninstantiated templates.

The linker doesn't even lift its finger.

Removing everything testing-related

The **DOCTEST_CONFIG_DISABLE** identifier affects all macros - asserts and logging macros are turned into a no-op with **((void)0)** - to require a semicolon - and subcases just vanish.

- It should be defined everywhere in a module (exe / dll)
- Compilation and linking become lightning fast
- Most of the test runner is also removed

Expression decomposition

```
CHECK(a == b);
```

Gets (sort of) expanded to:

```
do {
    ResultBuilder rb("CHECK", "main.cpp", 76, "a == b");
    try {
        rb.setResult(ExpressionDecomposer() << a == b);
    } catch(...) { rb.exceptionOccurred(); }
    if(rb.log()) // returns true if the assert failed
        BREAK_INTO_DEBUGGER();
} while((void)0, 0); // no "conditional expression is constant
```

In C++ the **<<** operator has higher precedence over **==**

That is how the decomposer captures the left operand "a".

Also the "Owl" technique (0,0) used to silence C4127 in MSVC

Expression decomposition

```
struct ExpressionDecomposer {
    template <typename L>
    LeftOperand<const L&> operator<<(const L& operand) {
        return LeftOperand<const L&>(operand);
    }
};
```

```
template <typename L>
struct LeftOperand{
    L lhs;
    LeftOperand(L in) : lhs(in) {}

    template <typename R> Result operator==(const R& rhs) {
        return Result(lhs == rhs, stringify(lhs, "==", rhs))
    }
};
```

Expression decomposition

```
struct Result {
    bool   passed;
    String decomposition;

    Result(bool p, const String& d) : passed(p) , decomposition(d) {}
};
```

```
template <typename L, typename R>
String stringify(const L& lhs, const char* op, const R& rhs) {
    return toString(lhs) + " " + op + " " + toString(rhs);
}
```

The default stringification of types is "**{?}**".

Translating exceptions


```
int func() { throw MyType(); return 0; }

CHECK(func() == 42);
```

```
main.cpp(34) ERROR!
  CHECK( func() == 42 )
threw exception:
  MyType: contents...
```

```
try {
    rb.setResult(ExpressionDecomposer() << func() == 42);
} catch(...) { rb.exceptionOccurred(); }
```

Translating exceptions


```
struct ITranslator { // interface
    virtual bool translate(String&) = 0;
};

template<typename T>
struct Translator : ITranslator {
    String(*m_func)(T); // function pointer
    Translator(String(*func)(T)) : m_func(func) {}

    bool translate(String& res) override {
        try {
            throw; // rethrow
        } catch(T ex) {
            res = m_func(ex); // use the translator
            return true;
        } catch(...) {
            return false; // didn't catch by T
        }
    }
};
```

Translating exceptions


```
REG_TRANSLATOR(const MyType& e) {
    return String("MyType: ") + toString(e);
}
```

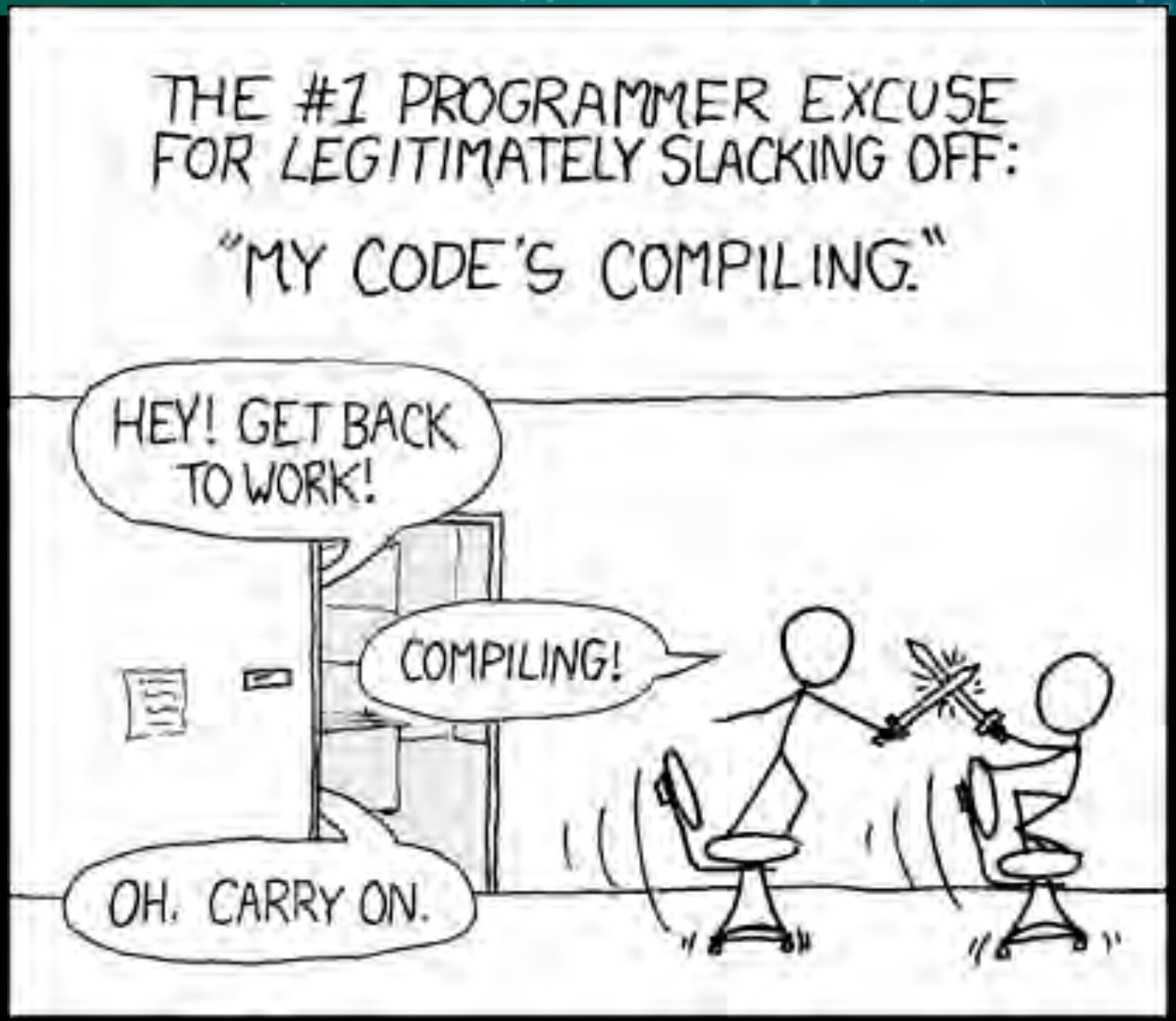
```
// REG_TRANSLATOR gets expanded to:
inline String ANON_TR_76(const MyType& e); // fwd decl
static int ANON_TR_77 = regTranslator(ANON_TR_76); // register
String ANON_TR_76(const MyType& e) {
    return String("MyType: ") + toString(e);
}
```

```
void reg_in_test_runner(ITranslator* t); // fwd decl

template<typename T>
int regTranslator(String(*func)(T)) {
    static Translator<T> t(func); // alive until the program ends
    reg_in_test_runner(&t);
    return 0;
}
```

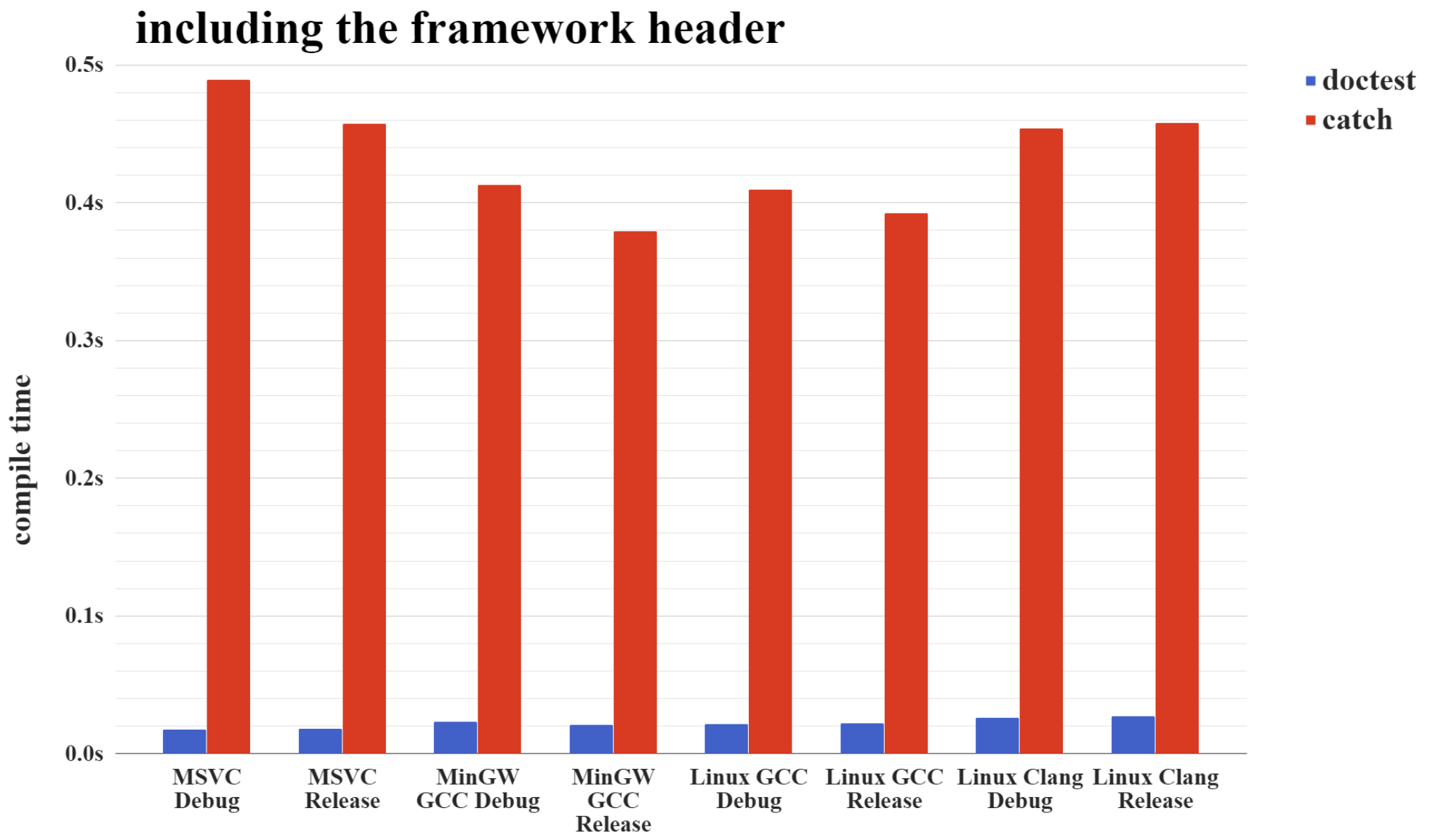
The Lippincott function


```
String translate() {
    // try translators
    String res;
    for(size_t i = 0; i < translators.size(); ++i)
        if(translators[i]->translate(res)) // if success
            return res;
    // proceed with default translation
    try {
        throw; // rethrow
    } catch(std::exception& ex) {
        return ex.what();
    } catch(std::string& msg) {
        return msg.c_str();
    } catch(const char* msg) {
        return msg;
    } catch(...) {
        return "Unknown exception!";
    }
}
void ResultBuilder::exceptionOccurred() { /* use translate() */ }
```

Major C++ issue - compile times



Compile times - header cost



Compile times - header cost

The doctest header is less than 1200 lines of code after the MSVC preprocessor (whitespace removed)

compared to 41k for Catch - 1.4 MB (Catch2 is 36k - 1.3 MB)

This is because doctest doesn't include anything in its forward declaration part.

The idea is not to bash Catch - it's an amazing project that continues to evolve (now Catch2) and deserves its reputation.

Using Boost.Test in its single header form is **A LOT** slower...

Forward declaring std::ostream

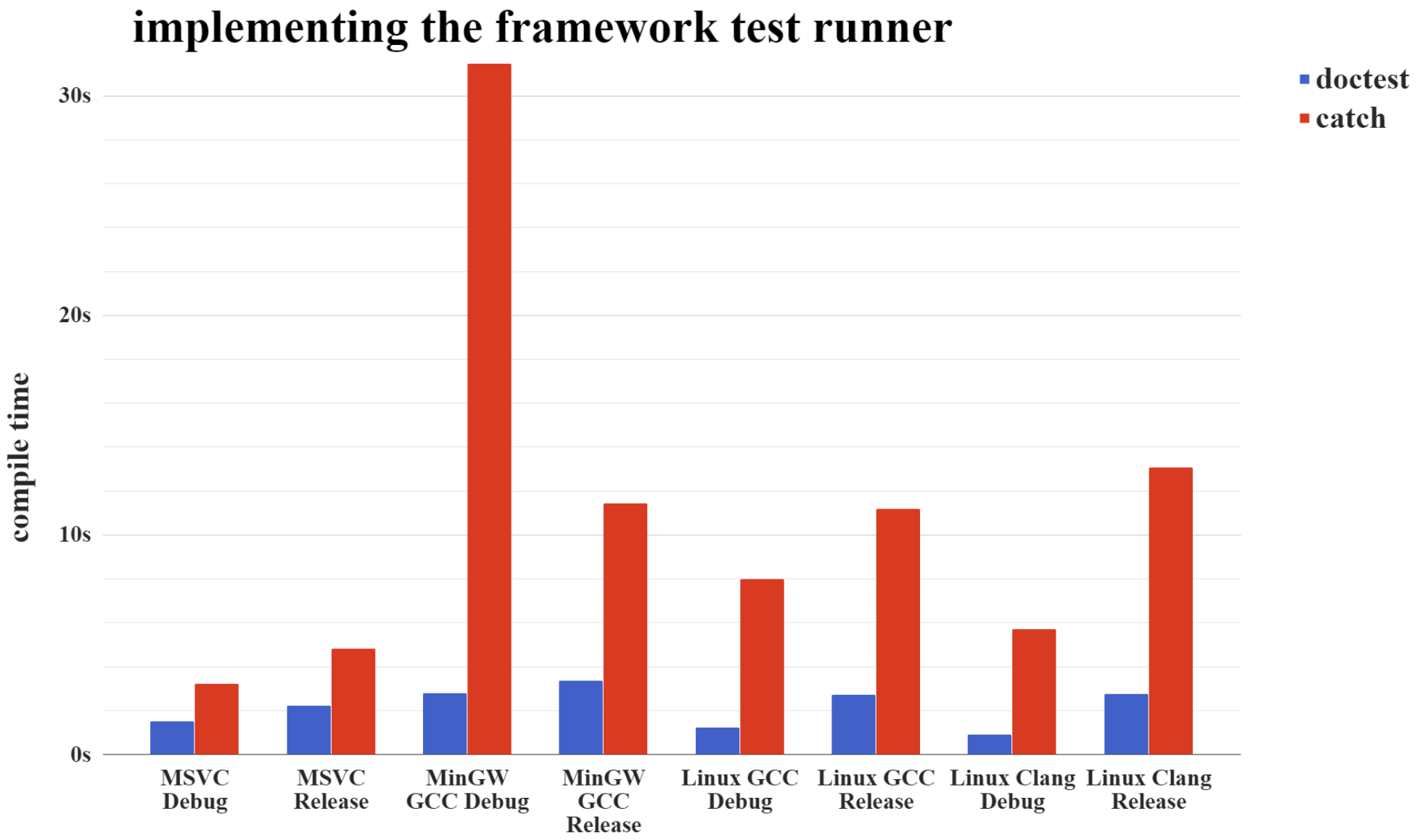
```
namespace std // forbidden by the standard but works like a charm
{
    template <class charT>                          struct char_traits;
    template <>                                     struct char_traits<char>;
    template <class charT, class traits>   class basic_ostream;
    typedef basic_ostream<char, char_traits<char> > ostream;
}
```

This is how the **doctest** header doesn't need to include headers for **std::nullptr_t** or **std::ostream**.

Just including the **<iosfwd>** header with MSVC leads to 9k lines of code after the preprocessor - 450kb...

Boost.DI does the same - forward declares stuff from std and doesn't include anything

Compile times - implementation cost



Compile times - assert macros

doctest normal	`CHECK(a == b);`
doctest binary	`CHECK_EQ(a, b); // no expression decomposition`
doctest fast	`FAST_CHECK_EQ(a, b); // not evaluated in a try {} block`
doctest faster	`FAST_CHECK_EQ(a, b); // DOCTEST_CONFIG_SUPER_FAST_ASSERTS`
catch normal	`CHECK(a == b);`
catch faster	`CHECK(a == b); // CATCH_CONFIG_FAST_COMPILE`

500 test cases with 100 asserts in each - 50k **CHECK(a==b)**