DOCTEST

The lightest feature-rich **C++** single-header testing framework for unit tests and TDD

Inspired by ***unittest {}*** from **D** and **Python**'s **docstrings/doctests**

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

A complete example with a self-registering test that compiles to an executable

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

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

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

and the output of that executable

```
[doctest] doctest version is "1.0.0"
[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 |    0 skipped
[doctest] assertions:    4 |    3 passed |    1 failed |
```

exit code != 0 because an assertion failed

Interface and functionality modeled mainly after **Catch**

Currently a few (big) things which Catch has are missing but **doctest** will eventually become a superset of Catch

Some ideas taken (or added to the roadmap) from **Boost.Test**, **googletest** and others

Distributed as a single header for simple integration

WHAT MAKES IT DIFFERENT

In 2 words: light and unintrusive

- ultra light on compile times for including the header
- can remove everything testing-related from the binary
- doesn't pollute the global namespace (or uses a prefix)
- all macros are (or can be) prefixed
- doesn't drag any headers with it (except where implemented)
- 0 warnings even on the most aggressive levels
- very portable and well tested C++98
- the user can easily provide the ***main()*** entry point
- can set options procedurally and not deal with argc/argv
- command line options can be prefixed to not clash with user

---

Unnoticeable even if included in every source file of your 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** (Linux/OSX)
- **Clang**: **3.4**, **3.5**, **3.6**, **3.7**, **3.8** (Linux/OSX)
- **MSVC**: **2008**, **2010**, **2012**, **2013**, **2015** (even **VC++6** from 1998!)
- warnings as errors even on the most aggressive warning levels
- output compared to one from a previous known good run
- ran through **valgrind** (Linux/OSX)
- ran through **address** and **UB** sanitizers (Linux/OSX)

---

a total of 220+ different configurations are built and tested

leveraging the free **travis** and **appveyor** CI services which are integrated with github

ALL THIS MAKES WRITING TESTS IN THE PRODUCTION CODE FEASIBLE! - LEADING TO:

- lower barrier for writing tests (no separate .cpp files)
- tests can be viewed as up-to-date comments
- tests can be optionally shipped to the customer for diagnosing bugs
- **TDD** in C++ has never been easier!

The library can be used like any other even if you don't like the idea of mixing production code and tests

MOST NOTABLE FEATURES OTHER THAN BEING LIGHT, TRANSPARENT AND STABLE

- only one core assertion macro for comparisons
- automatically registered tests
- subcases for shared setup/teardown between tests
- assertions for dealing with exceptions
- floating point comparison support - see the Approx() helper
- powerful mechanism for stringification of user types
- powerful command line with lots of options
- tests can be filtered by name/file/test suite using wildcards
- failures can break into the debugger on Windows and Mac
- colored output in the console
- can write tests in headers and still be registered only once
- range-based execution of tests

LET'S GET INTO DETAILS

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 <cstdio>
// test runner stuff...

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

UNIQUE ANONYMOUS VARIABLES

```
#define DOCTEST_CONCAT_IMPL(s1, s2) s1##s2
#define DOCTEST_CONCAT(s1, s2) DOCTEST_CONCAT_IMPL(s1, s2)
#ifdef __COUNTER__ // not standard and may be missing for some compilers
#define DOCTEST_ANONYMOUS(x) DOCTEST_CONCAT(x, __COUNTER__)
#else // __COUNTER__
#define DOCTEST_ANONYMOUS(x) DOCTEST_CONCAT(x, __LINE__)
#endif // __COUNTER__

int DOCTEST_ANONYMOUS(DOCTEST_ANON_VAR_); // int DOCTEST_ANON_VAR_5;
```

AUTO REGISTRATION

```
TEST_CASE("testing stuff") {
    // asserts
}
```

gets expanded to

```
static void DOCTEST_ANON_FUNC_1324();
static int DOCTEST_ANON_VAR_1325 = regTest(
    DOCTEST_ANON_FUNC_1324, "main.cpp", 56, "testing stuff");
void DOCTEST_ANON_FUNC_1324() {
    // asserts
}
```

static to not clash during linking with symbols with the same names from another translation unit

AUTO REGISTRATION

this resides in the test runner

```
std::set<Test>& getTestRegistry() { // return by reference
    static std::set<Test> data;     // important to be wrapped in a function
    return data;
}

typedef void (*funcType)(void);

int regTest(funcType f, const char* file, unsigned line, const char* name) {
    getTestRegistry().insert(Test(currentTestSuite(), name, f, file, line));
    return 0;
}
```

but the TEST_CASE macro produces warnings with clang!

-Wglobal-constructors

LETS TALK ABOUT WARNINGS

- -Weverything for Clang
- /W4 for MSVC (/Wall is madness - even std::vector produces thousands of warnings)
- -Wall -Wextra -pedantic for GCC (and over 50 other unique flags not covered by these! took a lot of time to find them)
- the full set of GCC warnings **https://github.com/Barro/compiler-warnings**

THE ADDITIONAL GCC FLAGS

-ansi -fstrict-aliasing -fstack-protector-all -funsafe-loop-optimizations -fdiagnostics-show-option -Wconversion -Wno-missing-field-initializers -Wold-style-cast -Wfloat-equal -Wlogical-op -Wundef -Wredundant-decls -Wshadow -Wstrict-overflow=5 -Wwrite-strings -Wpointer-arith -Wcast-qual -Wformat=2 -Wswitch-default -Wmissing-include-dirs -Wcast-align -Wformat-nonliteral -Wparentheses -Winit-self -Wuninitialized -Wswitch-enum -Wno-endif-labels -Wunused-function -Wnon-virtual-dtor -Wno-pmf-conversions -Wctor-dtor-privacy -Wsign-promo -Wsign-conversion -Wdisabled-optimization -Weffc++ -Winline -Winvalid-pch -Wstack-protector -Wunsafe-loop-optimizations -Wmissing-declarations -Woverloaded-virtual -Wstrict-null-sentinel -Wnoexcept -Wdouble-promotion -Wtrampolines -Wzero-as-null-pointer-constant -Wuseless-cast -Wvector-operation-performance -Wsized-deallocation -Wshift-overflow=2 -Wnull-dereference -Wduplicated-cond -Wmisleading-indentation -Wshift-negative-value

SILENCING WARNINGS IN THE HEADER

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

// ... header stuff

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

every (decent) compiler can do this

SILENCING WARNINGS IN MACROS

```
#ifdef __clang__
#define DOCTEST_REGISTER_FUNCTION(f, name)                                      \
    _Pragma("clang diagnostic push")                                            \
    _Pragma("clang diagnostic ignored \"-Wglobal-constructors\"")               \
    static int DOCTEST_ANONYMOUS(DOCTEST_ANON_VAR_) =                           \
                regTest(f, __LINE__, __FILE__, name);                           \
    _Pragma("clang diagnostic pop")
#endif // __clang__

#define DOCTEST_CREATE_AND_REGISTER(f, name)                                    \
    static void f();                                                            \
    DOCTEST_REGISTER_FUNCTION(f, name)                                          \
    inline void f()

#define DOCTEST_TEST_CASE(name)                                                 \
    DOCTEST_CREATE_AND_REGISTER(DOCTEST_ANONYMOUS(DOCTEST_ANON_FUNC_), name)
```

"_Pragma()" was standardized in C++11 but compilers support it for many years ("__pragma()" for MSVC) and since it's in the preprocessor "-std=c++98" doesn't bother us

but GCC also gives a warning - that the dummy int is not used

so... _Pragma() to the rescue? :)

NOOOOOT

_Pragma() in the C++ frontend of GCC (g++) isn't working in macros for quite some time (4+ years)

- **https://gcc.gnu.org/bugzilla/show_bug.cgi?id=55578**
- **https://gcc.gnu.org/bugzilla/show_bug.cgi?id=69543**

and luckily I can solve the warning with

```
__attribute__((unused))
```

```
#if defined(__GNUC__) && !defined(__clang__)
#define DOCTEST_REGISTER_FUNCTION(f, name)                                      \
    static int DOCTEST_ANONYMOUS(DOCTEST_ANON_VAR_) __attribute__((unused)) \
        = regTest(f, __LINE__, __FILE__, name);
#endif // __GNUC__
```

so far I haven't been able to suppress only -Waggregate-return in the CHECK() marco but it is more than worthless in C++

It's a completely anachronistic warning, since its motivation was to support backwards compatibility with C compilers that did not allow returning structures. Those compilers are long dead and are no longer of practical concern.

so I disable it at the begining of my header and leave it unpopped (don't tell anyone!)

SUBCASES

Code

```
TEST_CASE("lots of nested subcases") {
    cout << endl << "root" << endl;
    SUBCASE("") {
        cout << "1" << endl;
        SUBCASE("") { cout << "1.1" << endl; }
    }
    SUBCASE("") {
        cout << "2" << endl;
        SUBCASE("") { cout << "2.1" << endl; }
        SUBCASE("") {
            cout << "2.2" << endl;
            SUBCASE("") {
                cout << "2.2.1" << endl;
                SUBCASE("") { cout << "2.2.1.1" << endl; }
                SUBCASE("") { cout << "2.2.1.2" << endl; }
            }
        }
        SUBCASE("") { cout << "2.3" << endl; }
    }
}
```

Output

```
root
1
1.1

root
2
2.1

root
2
2.2
2.2.1
2.2.1.1

root
2
2.2
2.2.1
2.2.1.2

root
2
2.3
```

SUBCASE MACRO EXPANSION

```
#define DOCTEST_SUBCASE(name)                                           \
    if(const Subcase& DOCTEST_ANONYMOUS(DOCTEST_ANON_SUBCASE_) =   \
        Subcase(name, __FILE__, __LINE__))
```

```
SUBCASE("foo") {
    // some code in here
}
```

gets expanded to

```
if(const Subcase& DOCTEST_ANON_SUBCASE_54 = Subcase("foo", "main.cpp", 54)) {
    // some code in here
}
```

And the magic happens in the ctor/dtor of the ***Subcase*** class - each subcase is uniquely identified by the file and line

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;
    context.setOption("abort-after", 5);  // stop after 5 failed assertions
    context.applyCommandLine(argc, argv);
    context.setOption("no-breaks", true); // don't break in the debugger
    int res = context.run(); // run queries or run tests unless with --no-run
    if(context.shouldExit()) // query flags (and --exit) rely on you doing this
        return res;          // propagate the result of the tests
    // your program
    return res; // + your_program_res
}
```

designed for easy interop with the host application

REMOVING EVERYTHING TESTING-RELATED FROM THE BINARY

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

```
#define DOCTEST_CREATE_AND_REGISTER_FUNCTION(f, name) \
    template <typename T>                             \
    static inline void f()

#define DOCTEST_TEST_CASE(name)                       \
    DOCTEST_CREATE_AND_REGISTER_FUNCTION(             \
        DOCTEST_ANONYMOUS(DOCTEST_ANON_FUNC_), name)
```

so all test cases are turned into uninstantiated templates

the linker doesn't even lift his finger

The ***DOCTEST_CONFIG_DISABLE*** identifier affects all macros - assertions are turned into a noop using ***((void)0)*** and subcases just vanish - leaving only the ***{}*** code block.

It should be defined everywhere in a module (exe/dll)

This makes compilation and linking lightning fast - almost like the tests don't exist

Most of the test runner is also removed

ASSERTION MACROS

3 levels - ***WARN***, ***CHECK*** and ***REQUIRE***

- ***WARN*** - doesn't fail the test case but prints a message
- ***CHECK*** - fails the test case and prints a message but continues
- ***REQUIRE*** - fails the test case, prints and ends it immediately

```
WARN(true == false); // just a message
CHECK(1 < 0);        // will fail the test case but continue
REQUIRE(4 == 8);     // will end the test case
REQUIRE(1 != 2);     // never reached
```

a standard C++ operator for the comparison is used - yet the full expression is decomposed and the left/right values are logged

exceptions used for ***REQUIRE*** to terminate the current test case

EXPRESSION DECOMPOSITION

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

gets expanded to

```
do {
    ResultBuilder rb("CHECK", "main.cpp", 76, "a == b");
    try {
        rb.setResult(ExpressionDecomposer() << a == b);
    } catch(...) { rb.m_threw = true; }
    if(rb.log()) // returns true if the expression is false (or threw)
        DOCTEST_BREAK_INTO_DEBUGGER(); // a macro
    rb.react(); // for REQUIRE macros will throw an exception
} while(always_false());
```

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

And that is how the expression decomposer captures the lhs

EXPRESSION DECOMPOSITION

```
struct ExpressionDecomposer {
    template <typename L>
    Expression_lhs<const L&> operator<<(const L& operand) {
        return Expression_lhs<const L&>(operand); // returns a different type
    }
};
```

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

    // if not a binary expression
    operator Result() { return Result(!!lhs, toString(lhs)); }

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

EXPRESSION DECOMPOSITION

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

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

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

STRINGIFICATION OF TYPES

```
template <typename T>
String toString(const T& value) {
    return StringMaker<T>::convert(value);
}

String toString(const char* in);
String toString(bool in);
String toString(float in);
String toString(double in);
String toString(char in);
String toString(char unsigned in);
String toString(int in);
String toString(int unsigned in);
```

toString() is the root of the stringification chain

STRINGIFICATION OF TYPES

```
template <bool C> struct StringMakerBase {
    template <typename T>
    static String convert(const T&) { return "{?}"; } // default
};
template <> struct StringMakerBase<true> {
    template <typename T>
    static String convert(const T& in) {
        std::ostream* stream = createStream();
        *stream << in;
        String result = getStreamResult(stream);
        freeStream(stream);
        return result;
    }
};
template <typename T>
struct StringMaker : StringMakerBase<has_insertion_operator<T>::value> {};
```

```
std::ostream* createStream();                // defined in the test runner
String        getStreamResult(std::ostream*); // defined in the test runner
void          freeStream(std::ostream*);     // defined in the test runner
```

note that the operations with the stream use a pointer

(operator<< takes a reference which is basically a pointer)

THE HAS_INSERTION_OPERATOR TRAIT

```
typedef char no;
typedef char yes[2];

struct any_t {
    template <typename T>
    any_t(const T&);
};

yes& testStreamable(std::ostream&);
no   testStreamable(no);

no operator<<(const std::ostream&, const any_t&);
```

```
template <typename T>
struct has_insertion_operator {
    static std::ostream& s;
    static const T&      t;
    static const bool value = sizeof(testStreamable(s << t)) == sizeof(yes);
};
```

```
bool is_int_ostream_streamable = has_insertion_operator<int>::value;
```

FORWARD DECLARING STD::OSTREAM

```
#ifdef __clang__
    #include <ciso646> // to detect if libc++ is being used with clang
#endif // __clang__
```

```
#ifdef _LIBCPP_VERSION
    // forward declaration was troublesome and <iosfwd> is very light in libc++
    #include <iosfwd>
#else // _LIBCPP_VERSION
    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;
    }
#endif // _LIBCPP_VERSION
```

If the user wants to be pedantic - there is a configuration identifier - ***DOCTEST_CONFIG_USE_IOSFWD*** - that forces doctest to include "iosfwd" and not forward declare types from std

STRINGIFICATION OF TYPES

- An overload of "toString()" can be provided for user types
- Types that are "std::ostream" streamable work out of the box
- Using the "StringMaker" class allows for partial template specialization and thus a container like "std::vector" can be stringified easily - without being concrete about the allocator
- The default stringification of types is "{?}"

A lot of effort went into:

- not dragging any headers (except in the implementation file)
  - this meant implementing my own String class
- warning-free 220+ different builds running cleanly through valgrind/sanitizers (more than 100 hours went into this)

The usual suspects (problematic warnings) are:

- -Winline
- -Weffc++
- -Wstrict-overflow

A valgrind error took me 3-4 days to track it down - only with g++4.8 and only in release - strcmp() was reading strings in chunks of 4 bytes for speed and if a string didn't have a length multiple of 4 - valgrind complained

Hit MANY other toolchain problems

BENCHMARKS

- the doctest header is ~50 times lighter on compile times for inclusion compared to Catch (<10ms compared to 450ms)
- the doctest header is less than 600 lines of code after the preprocessor - compared to the 400k of Catch
- just including the iosfwd header with MSVC leads to ~40k lines of code after the preprocessor compared to the 200 of clang with libc++

ROADMAP

- improved startup time - less allocations
- a mechanism for translating user defined exceptions
- crash handling: signals on UNIX (and SE on Windows)
- support for tagging - separate from test case names
- reporters - xml, jUnit/xUnit and user defined
- convolution support for the assertion macros
- time stuff - how much ms does a test take
- add contextual info to asserts - with an INFO macro
- running tests a few times
- test execution in separate processes - UNIX fork()
- detect floating point exceptions

MORE ROADMAP!

- generators
- matchers
- customizing the colors in the console output
- utf8?
- wchar?
- allocator for String class
- MSTest integration (and also maybe with XCode)
- and many many other small things!

And all this without having ever written or dealt with unit tests (still haven't)

https://github.com/onqtam/doctest

This presentation was made possible **NO** thanks to ***Dota 2***

That cancer tried it's best but I defeated it 2 days ago (deleted)

http://onqtam.github.io/slides/doctest.html

http://onqtam.github.io/

https://github.com/onqtam

https://www.facebook.com/viktor.i.kirilov

https://stackoverflow.com/users/3162383/onqtam

https://www.linkedin.com/in/viktor-kirilov-98409a29

https://twitter.com/KirilovVik

vik.kirilov@gmail.com

Q&A