

# Clean Test

A modern testing framework

---

Philipp Ochsendorf

November 17th, 2022



We are hiring!

- Meaningful work:  
we break down language barriers
- Fast growing European startup
- Interesting challenges at scale
- Modern C++, latest tools
- Work independently with smart colleagues
- Many open positions in engineering teams,  
several with a C++ focus (backend, research)

# Clean Test: a modern testing framework

- Standard C++20, no macros
- Parallel test execution, thread safe expectations
- CI ready: JUnit export, UTF-8 support
- CMake library without dependencies
- Liberal BSL-1.0 license

```
1 #include <clean-test/clean-test.h>
2
3 namespace ct = clean_test;
4 using namespace ct::literals;
5
6 auto const s0 = "math"_suite = [] {
7     "integer"_test = [] {
8         ct::expect(1_i + 2 == 3);
9     };
10 };
```

# Suites and Tests

```
1 #include <clean-test/clean-test.h>
2 #include <iostream>
3
4 namespace ct = clean_test;
5 using namespace ct::literals;
6
7 auto const talk = "talk"_suite = [] {
8     "hello world"_test = [] {
9         // Your tests go here
10         std::cout << "Hello Meeting C++ 2022!\n";
11     };
12 };
```

**Suite** test initialization executed  
automatically during program start

**Tests** standalone callable performing  
runtime checks



# Suites and Tests

```
1 #include <clean-test/clean-test.h>
2 #include <iostream>
3
4 namespace ct = clean_test;
5 using namespace ct::literals;
6
7 auto const talk = "talk"_suite = [] {
8     "hello world"_test = [] {
9         // Your tests go here
10         std::cout << "Hello Meeting C++ 2022!\n";
11     };
12 };
13
14 auto const intro = ct::Suite{"intro"} = [] {
15     ct::Test{"demo", [] {
16         std::cout << "Hi again\n";
17     }};
18 };
```

**Suite** test initialization executed  
automatically during program start

**Tests** standalone callable performing  
runtime checks



# Suites and Tests

```
1 #include <clean-test/clean-test.h>
2 #include <iostream>
3
4 namespace ct = clean_test;
5 using namespace ct::literals;
6
7 auto const talk = "talk"_suite = [] {
8     "hello world"_test = [] {
9         // Your tests go here
10         std::cout << "Hello Meeting C++ 2022!\n";
11     };
12 };
13
14 auto const intro = ct::Suite{"intro"} = [] {
15     ct::Test{"demo", [] {
16         std::cout << "Hi again\n";
17     }};
18
19     "unstable"_tag / "annoying"_test = [] {
20         std::cout << "Hi once more.\n";
21     };
22 };
```

- Suite** test initialization executed automatically during program start
- Tests** standalone callable performing runtime checks



# Runtime Configurable

```
> ./demo
```

```
C1 [ ===== ] Running 3 test-cases
C2 [ RUN      ] intro/demo
C3 [ RUN      ] talk/hello world
C4 Hello Meeting C++ 2022!
C5 Hi again
C6 [ RUN      ] intro/annoying
C7 Hi once more.
C8 [ PASS     ] intro/demo (5.417us)
C9 [ PASS     ] talk/hello world (748.0ns)
C10 [ PASS     ] intro/annoying (803.0ns)
C11 [ ===== ] Ran 3 test-cases (729.3us total)
C12 [ PASS     ] All 3 test-cases
```

- Test selection
- Controlling parallelism
- Further options with `--help`

```
> ./demo --list
```

```
C1 | intro
C2 |   | annoying {unstable}
C3 |   | demo
C4 | | talk
C5 |   | hello world
```

# Expectations

```
1 #include <clean-test/clean-test.h>
2
3 namespace ct = clean_test;
4 using namespace ct::literals;
5
6 auto const t = "expect"_test = [] {
7     ct::expect(true);
8     ct::expect(0 == 7);
9 };
```

```
C1 [ ===== ] Running 1 test-cases
C2 [ RUN      ] expect
C3 Failure in demo.cpp:8
C4 false
C5 [ FAIL     ] expect (545.9us)
C6 [ ===== ] Ran 1 test-cases (1.324ms total)
C7 [ FAIL     ] expect
```



# Expectations with Expression Introspection

- `ct::lift` wraps anything for introspection
- Many UDLs for lifting scalar and string types

```
1 #include <clean-test/clean-test.h>
2
3 namespace ct = clean_test;
4 using namespace ct::literals;
5
6 auto const t = "expect"_test = [] {
7     auto v = 1;
8     auto * p = &v;
9     ct::expect(nullptr == ct::lift(p));
10    ct::expect(v == 7_i);
11 };
```

```
C1 [ ===== ] Running 1 test-cases
C2 [ RUN      ] expect
C3 Failure in demo.cpp:9
C4 ( nullptr == 0x7f51cb6e38ac )
C5 Failure in demo.cpp:10
C6 ( 1 == 7 )
C7 [ FAIL     ] expect (30.50us)
C8 [ ===== ] Ran 1 test-cases (1.253ms total)
C9 [ FAIL     ] expect
```

- Implemented with *Expression Templates* [T. Veldhuizen 1995, D. Vandevorode 2002]

## Expression Introspection Attempt (1/4)

```
6 struct Tag{};
7
8 template <typename T>
9 class Clause final : public Tag {
10     T m_value;
11
12 public:
13     template <typename U>
14     constexpr Clause(U && value)
15         : m_value{std::forward<U>(value)} {}
16
17     constexpr decltype(auto) value() const {
18         return m_value;
19     }
20
21     constexpr explicit operator bool() const {
22         return static_cast<bool>(value());
23     }
24
25     friend std::ostream & operator<<(
26         std::ostream & out, Clause const & clause) {
27         return out << clause.value();
28     }
29 };
```

```
31 template <typename T>
32 concept Expression =
33     std::derived_from<std::remove_cvref_t<T>, Tag>;
34
35 template <typename T>
36 constexpr decltype(auto) lift(T && t) {
37     if constexpr (Expression<T>) {
38         return std::forward<T>(t);
39     } else {
40         return Clause<T>{std::forward<T>(t)};
41     }
42 }
```

## Expression Introspection Attempt (2/4)

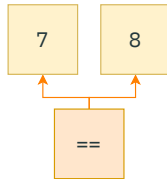
```
46 template <Expression Lhs, Expression Rhs>
47 class Equal final : public Tag {
48     Lhs m_lhs;
49     Rhs m_rhs;
50
51 public:
52     constexpr Equal(auto && lhs, auto && rhs)
53     : m_lhs{std::forward<decltype(lhs)>(lhs)},
54       m_rhs{std::forward<decltype(rhs)>(rhs)} {}
55
56     constexpr auto value() const {
57         return m_lhs.value() == m_rhs.value();
58     }
59
60     constexpr explicit operator bool() const {
61         return static_cast<bool>(value());
62     }
63
64     friend std::ostream & operator<<(
65         std::ostream & out, Equal const & eq) {
66         return out << "(" << eq.m_lhs
67            << " == " << eq.m_rhs << " )";
68     }
69 };
```

```
71 template <typename Lhs, typename Rhs>
72 Equal(Lhs &&, Rhs &&) -> Equal<Lhs, Rhs>;
73
74 template <typename Lhs, typename Rhs>
75 constexpr auto operator==(Lhs && lhs, Rhs && rhs) {
76     return Equal{
77         lift(std::forward<Lhs>(lhs)),
78         lift(std::forward<Rhs>(rhs)),
79     };
80 }
```

## Expression Introspection Attempt (3/4)

```
82 template <typename T>
83 void expect(T && t) {
84     using Tags = std::array<std::string_view, 2>;
85     static auto const tags = Tags{"FAIL", "OK "};
86
87     std::cout
88         << tags[static_cast<bool>(t)]
89         << ": " << t << std::endl;
90 }
91
94 int main() {
95     static_assert(7 == ct::lift(7));
96     ct::expect(7 == ct::lift(8));
97 }
```

c1 FAIL: ( 7 == 8 )



# Required Operators

## Overloaded Operators

Unary      `not`, `+`, `-`, `*` (dereference)

Arithmetic      `+`, `-`, `*`, `/`, `%`

Bitwise      `&`, `|`, `~`, `^`

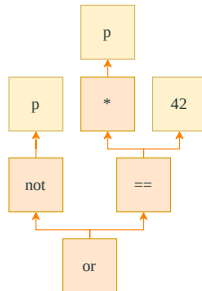
Comparison      `<`, `<=`, `>`, `>=`, `==`, `!=`

Special      `,` (comma), `and`, `or`

## Expression Introspection Attempt (4/4)

```
92 template <Expression Lhs, Expression Rhs>
93 class Or final : public Tag {
94     Lhs m_lhs;
95     Rhs m_rhs;
96
97 public:
98     constexpr Or(auto && lhs, auto && rhs)
99     : m_lhs{std::forward<decltype(lhs)>(lhs)},
100       m_rhs{std::forward<decltype(rhs)>(rhs)} {}
101
102     constexpr decltype(auto) value() const {
103         return m_lhs.value() or m_rhs.value();
104     }
105
106     constexpr explicit operator bool() const {
107         return static_cast<bool>(value());
108     }
109
110     friend std::ostream & operator<<(
111         std::ostream & out, Or const & o) {
112         return out << "(" << o.m_lhs
113            << " or " << o.m_rhs << " )";
114     }
115 };
```

```
184 int main() {
185     static_assert(7 == ct::lift(7));
186     ct::expect(7 == ct::lift(8));
187
188     int const * const p = nullptr;
189     ct::expect(
190         not ct::lift(p) or *ct::lift(p) == 42);
191 }
```

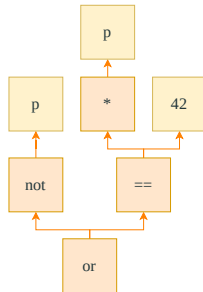


## Expression Introspection Attempt (4/4)

```
92 template <Expression Lhs, Expression Rhs>
93 class Or final : public Tag {
94     Lhs m_lhs;
95     Rhs m_rhs;
96
97 public:
98     constexpr Or(auto && lhs, auto && rhs)
99     : m_lhs{std::forward<decltype(lhs)>(lhs)},
100       m_rhs{std::forward<decltype(rhs)>(rhs)} {}
101
102     constexpr decltype(auto) value() const {
103         return m_lhs.value() or m_rhs.value();
104     }
105
106     constexpr explicit operator bool() const {
107         return static_cast<bool>(value());
108     }
109
110     friend std::ostream & operator<<(
111         std::ostream & out, Or const & o) {
112         return out << "(" << o.m_lhs
113            << " or " << o.m_rhs << " )";
114     }
115 };
```

```
184 int main() {
185     static_assert(7 == ct::lift(7));
186     ct::expect(7 == ct::lift(8));
187
188     int const * const p = nullptr;
189     ct::expect(
190         not ct::lift(p) or *ct::lift(p) == 42);
191 }
```

C1 FAIL: ( 7 == 8 )  
C2 segmentation fault (core dumped)



## Expression Introspection Cached Attempt (1/2)

```
92 template <typename Expression>
93 using Value =
94     decltype(std::declval<Expression>().value());
95
96 template <Expression Lhs, Expression Rhs>
97 class CachingOr final : public Tag {
98     Lhs m_lhs;
99     Rhs m_rhs;
100     Value<Lhs> m_lhs_value;
101     std::optional<Value<Rhs>> m_rhs_value;
102
103 public:
104     constexpr CachingOr(auto && lhs, auto && rhs)
105     : m_lhs{std::forward<decltype(lhs)>(lhs)},
106       m_rhs{std::forward<decltype(rhs)>(rhs)},
107       m_lhs_value{m_lhs.value()},
108       m_rhs_value{m_lhs_value
109                   ? std::nullopt
110                   : std::make_optional(m_rhs.value())}
111     {}
112
113     constexpr decltype(auto) value() const {
114         return m_lhs_value or *m_rhs_value;
115     }
116
117     constexpr explicit operator bool() const {
118         return static_cast<bool>(value());
119     }
120
121     friend std::ostream & operator<<(
122         std::ostream & out, CachingOr const & o)
123     {
124         out << "(" << o.m_lhs << " or ";
125         if (o.m_rhs_value) {
126             out << o.m_rhs;
127         } else {
128             out << "<unknown>";
129         }
130         return out << " ";
131     }
132 };
```



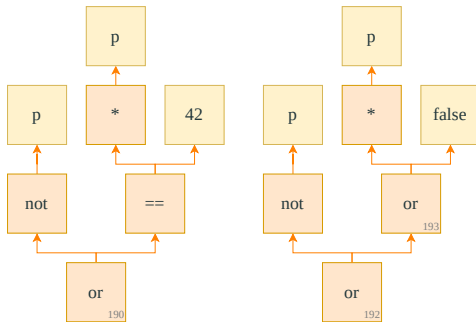
## Expression Introspection Cached Attempt (2/2)

```
184 int main() {  
185     static_assert(7 == ct::lift(7));  
186     ct::expect(7 == ct::lift(8));  
187  
188     int const * const p = nullptr;  
189     ct::expect(  
190         not ct::lift(p) or *ct::lift(p) == 42);  
191 }
```

```
C1 FAIL: ( 7 == 8 )  
C2 OK   : ( not 0 or <unknown> )
```

## Expression Introspection Cached Attempt (2/2)

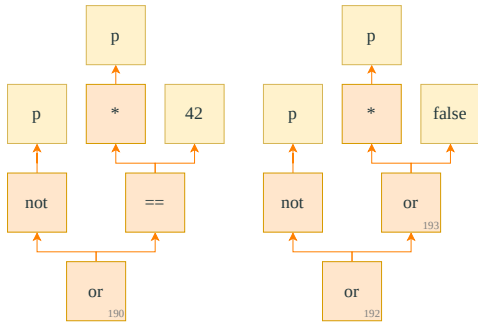
```
184 int main() {  
185     static_assert(7 == ct::lift(7));  
186     ct::expect(7 == ct::lift(8));  
187  
188     int const * const p = nullptr;  
189     ct::expect(  
190         not ct::lift(p) or *ct::lift(p) == 42);  
191     ct::expect(  
192         not ct::lift(p) or  
193         (*ct::lift(p) or false));  
194 }
```



```
C1 FAIL: ( 7 == 8 )  
C2 OK  : ( not 0 or <unknown> )  
C3 segmentation fault (core dumped)
```

## Expression Introspection Cached Attempt (2/2)

```
184 int main() {  
185     static_assert(7 == ct::lift(7));  
186     ct::expect(7 == ct::lift(8));  
187  
188     int const * const p = nullptr;  
189     ct::expect(  
190         not ct::lift(p) or *ct::lift(p) == 42);  
191     ct::expect(  
192         not ct::lift(p) or  
193         (*ct::lift(p) or false));  
194 }
```



```
C1 FAIL: ( 7 == 8 )  
C2 OK  : ( not 0 or <unknown> )  
C3 segmentation fault (core dumped)
```

- May only cache upon `value()` call
- Requires `mutable` members; not `constexpr`.

*All problems in computer science can be solved  
by another level of indirection.*

David J. Wheeler

## Expression Evaluation with Introspection (1/3)

```
176 template <Expression Lhs, Expression Rhs>
177 class Or final : public Tag {
178     Lhs m_lhs;
179     Rhs m_rhs;
180
181 public:
182     using Evaluation = OrEvaluation<typename Lhs::Evaluation, typename Rhs::Evaluation>;
183     friend Evaluation;
184
185     constexpr Or(auto && lhs, auto && rhs)
186     : m_lhs{std::forward<decltype(lhs)>(lhs)},
187       m_rhs{std::forward<decltype(rhs)>(rhs)}
188     {}
189
190     [[nodiscard]] constexpr auto evaluation() const {
191         return Evaluation{*this};
192     }
193
194     constexpr explicit operator bool() const {
195         return static_cast<bool>(evaluation().value());
196     }
197 };
```

## Expression Evaluation with Introspection (2/3)

```
137 template <typename Lhs, typename Rhs>
138 class OrEvaluation final {
139     Lhs m_lhs;
140     std::optional<Rhs> m_rhs;
141     bool m_value;
142
143 public:
144     OrEvaluation(auto && expr)
145     : m_lhs{expr.m_lhs.evaluation()},
146       m_rhs{
147           static_cast<bool>(m_lhs.value())
148           ? std::nullopt
149           : std::optional{
150               expr.m_rhs.evaluation()
151           }
152       },
153       m_value{
154           m_rhs
155           ? static_cast<bool>(m_rhs->value())
156           : static_cast<bool>(m_lhs.value())
157       }
158     {}
159
```

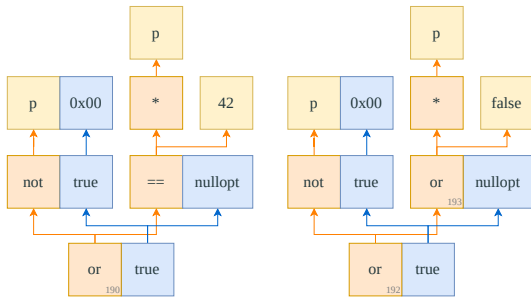
```
160     constexpr decltype(auto) value() const {
161         return m_value;
162     }
163
164     friend std::ostream & operator<<(
165         std::ostream & out, OrEvaluation const & eval) {
166         out << "(" << eval.m_lhs << " or ";
167         if (eval.m_rhs) {
168             out << *eval.m_rhs;
169         } else {
170             out << "<unknown>";
171         }
172         return out << " ";
173     }
174 };
```

# Expression Evaluation with Introspection (3/3)

```
61 template <Expression T>
62 void expect(T && t) {
63     using Tags = std::array<std::string, 2>;
64     static auto const tags
65         = Tags{"FAIL", "OK "};
66
67     auto const eval = t.evaluation();
68     std::cout
69         << tags[static_cast<bool>(eval.value())]
70         << ": " << eval << std::endl;
71 }
```

```
184 int main() {
185     static_assert(7 == ct::lift(7));
186     ct::expect(7 == ct::lift(8));
187
188     int const * const p = nullptr;
189     ct::expect(
190         not ct::lift(p) or *ct::lift(p) == 42);
191     ct::expect(
192         not ct::lift(p) or
193         (*ct::lift(p) or false));
194 }
```

```
C1 FAIL: ( 7 == 8 )
C2 OK  : ( not 0 or <unknown> )
C3 OK  : ( not 0 or <unknown> )
```



# Limitations of `ct::lift`

- Ensure that operators have at least one `ct::Expression` operand, e.g. `ct::lift`.
- Function calls are not lazy by default

```
15 int * p = nullptr;
16 ct::expect(not ct::lift(p) or *ct::lift(p)); // ok
17
18 ct::expect(not ct::lift(p) or use(*p)); // ERROR
19 // Alternative 1
20 if (p) {
21     ct::expect(use(*p));
22 }
23 // Alternative 2: lifting callable
24 ct::expect(not ct::lift(p) or ct::lift([&] { return use(*p); }));
```



# Advanced Expectations

- Debug output

```
22 ct::expect(false) << "but why?";
```

- Assertions: **asserted** / **asserted\_if**

```
26 ct::expect(false) << ct::asserted;  
27 ct::expect(1 / 0 == 42) << "not executed";
```

- Flakyness: **flaky** / **flaky\_if**

```
33 ct::expect(false) << ct::flaky_if(on_linux());
```

- Exceptions: **throws** / **throws<Exception>**

```
39 ct::expect(not ct::throws([] {}));
```

- Fatal Assertions: **aborts** / **debug\_aborts**

```
49 ct::expect(ct::debug_aborts([] {  
50     assert(false);  
51 }));
```

```
C1 [ ===== ] Running 5 test-cases  
C2 [ RUN      ] advanced/debug output  
C3 Failure in demo.cpp:22  
C4 false  
C5 but why?  
C6 [ FAIL     ] advanced/debug output (30.50us)  
C7 [ RUN      ] advanced/asserted  
C8 Failure (asserted) in demo.cpp:26  
C9 false  
C10 [ ABORT    ] advanced/asserted (25.23us)  
C11 [ RUN      ] advanced/flaky  
C12 Failure (flaky) in demo.cpp:33  
C13 false  
C14 [ PASS     ] advanced/flaky (4.476us)  
C15 [ RUN      ] advanced/throw  
C16 [ PASS     ] advanced/throw (14.88us)  
C17 [ RUN      ] advanced/abort  
C18 [ PASS     ] advanced/abort (207.4ms)  
C19 [ ===== ] Ran 5 test-cases (208.0ms total)  
C20 [ FAIL     ] advanced/debug output  
C21 [ FAIL     ] advanced/asserted  
C22 [ PASS     ] All other 3 test-cases
```

# Advanced Distance Checks

```
17 static_assert(ct::utils::norm(-2) == 2_i);
18
19 auto const x = 0.15_d + 0.15;
20 auto const y = 0.1_d + 0.2;
21
22 ct::expect(x == y);
23 ct::expect(ct::is_close(x, y));
24 ct::expect(ct::is_close(x, y + .5));
25
26 ct::expect(
27     ct::distance(x, y + .5) <= ct::tolerance(.5));
```

- CPO `ct::utils::norm`

- Absolute distance

$$\Delta_{\text{abs}}(x, y) = \|x - y\|$$

- Relative distance

$$\Delta_{\text{rel}}(x, y) = \frac{\|x - y\|}{\max(\|x\|, \|y\|)}$$

```
C1 [ ===== ] Running 1 test-cases
C2 [ RUN      ] distance/close
C3 Failure in demo.cpp:22
C4 ( ( 0.15 + 0.15 ) == ( 0.1 + 0.2 ) )
C5 Failure in demo.cpp:24
C6 ( distance(( 0.15 + 0.15 ), ( ( 0.1 + 0.2 ) + 0.5 )) = {absolute: 0.5, relative: 0.625}
C7 <= {absolute: 2.22045e-16, relative: 2.22045e-16} )
C8 [ FAIL     ] distance/close (116.9us)
C9 [ ===== ] Ran 1 test-cases (278.5us total)
C10 [ FAIL     ] distance/close
```

# Conversion Tools

## clean-test/migration

- Hackable converter for existing tests
- `ct::lift` expectations for introspection

### *Original*

```
1 #define BOOST_TEST_MAIN
2 #include <boost/test/unit_test.hpp>
3
4 BOOST_AUTO_TEST_SUITE(demo)
5
6 BOOST_AUTO_TEST_CASE(talk)
7 {
8     BOOST_WARN(2 + 2 == 2 * 2);
9     BOOST_CHECK_MESSAGE(
10         3 + 3 == 2 * 3, "can't touch this");
11     BOOST_REQUIRE_EQUAL(4 + 4, 2 * 4);
12 }
13
14 BOOST_AUTO_TEST_SUITE_END()
```

### *Converted*

```
1 #include <clean-test/clean-test.h>
2
3 namespace ct = clean_test;
4 using namespace ct::literals;
5
6 auto const demo = "demo"_suite = [] {
7
8     "talk"_test = []
9     {
10         ct::expect(2_i + 2 == 2_i * 2) << ct::flaky;
11         ct::expect(
12             3_i + 3 == 2_i * 3) << "can't touch this";
13         ct::expect(4_i + 4 == 2_i * 4) << ct::asserted;
14     };
15
16 };
```

# Concurrent Tests

- Test-case attribution managed via `ct::Observer`
- Automatic for single threaded tests
- Propagate `ct::Observer` for advanced parallel tests
- `ct::expect` thread safe

```
20 void async(auto run) { std::async(run).wait(); }
21
22 auto const t = "par"_test = [] (ct::Observer & o) {
23     ct::expect(true);
24     ct::expect(o, true);
25     async([&] {
26         ct::expect(o, true);
27         ct::expect(true); // WRONG
28     });
29     async([&] {
30         auto const setup = ct::ObservationSetup{o};
31         ct::expect(true); // now ok
32     });
33 };
```

```
C1 [ ==== ] Running 1 test-cases
C2 [ RUN   ] par
C3 [ PASS  ] par (363.1us)
C4 [ ----- ] Warning: Observed test-expectations at unknown Observer.
C5 [       ] - demo.cpp:27
C6 [ ----- ] This is likely caused by missing to propagate an Observer.
C7 [ ==== ] Ran 1 test-cases (1.191ms total)
C8 [ PASS  ] All 1 test-cases
```

# Data Driven Tests

```
15 auto str(auto && v) {
16     auto buffer = std::ostringstream{};
17     buffer << std::forward<decltype(v)>(v);
18     return std::move(buffer).str();
19 }
20
21 auto const s = "data"_suite = [] {
22     static auto const data = std::vector{0, 1337};
23     ct::Test{"static", data} = [](int const n) {
24         ct::expect(n > 0_i);
25     };
26
27     std::tuple{0, "1337"}
28     | "temporary"_test = [](auto v) {
29         ct::expect(str(v) != "1337"_sv);
30     };
31 };
```

```
C1 [ ===== ] Running 4 test-cases
C2 [ RUN      ] data/static/0
C3 Failure in demo.cpp:24
C4 ( 0 > 0 )
C5 [ FAIL     ] data/static/0 (16.35us)
C6 [ RUN      ] data/static/1337
C7 [ PASS     ] data/static/1337 (7.029us)
C8 [ RUN      ] data/temporary/0
C9 [ PASS     ] data/temporary/0 (11.73us)
C10 [ RUN      ] data/temporary/1337
C11 Failure in demo.cpp:29
C12 ( "1337" != "1337" )
C13 [ FAIL     ] data/temporary/1337 (17.45us)
C14 [ ===== ] Ran 4 test-cases (294.3us total)
C15 [ FAIL     ] data/static/0
C16 [ FAIL     ] data/temporary/1337
C17 [ PASS     ] All other 2 test-cases
```

# Wrap-up









## Summary: Clean Test

- is a modern, versatile and yet simple to use testing framework.
- supports short-circuit expression introspection without macros.
- provides various productivity features.
- is built for parallel tests and test execution.

## Future Work

- Optimize test scheduling
- Lazy data providers
- Further migration utilities
- Convenience (e.g. for ranges)

## References

Clean Test  [clean-test/clean-test](#)  
Migrate  [clean-test/migration](#)  
This Talk  [clean-test/talk](#)  
Philipp      [@m8mble](#)

Comments, issues and PRs welcome.

# Asio Thread Pool

```
11 class AsioPool {
12     asio::io_context m_context;
13     std::optional<asio::io_context::work> m_work;
14     std::vector<std::jthread> m_workers;
15
16 public:
17     AsioPool(
18         ct::Observer & o,
19         std::size_t const n)
20     : m_context{},
21       m_work{m_context}
22     {
23         while(m_workers.size() < n) {
24             m_workers.emplace_back([&, this] {
25                 auto const os = ct::ObservationSetup{o};
26                 m_context.run();
27             });
28         }
29     }
30
31     ~AsioPool() {
32         m_work.reset();
33     }
```

```
35     auto executor() {
36         return m_context.get_executor();
37     }
38 };
39
40
41 auto test = "test"_test = [] (ct::Observer & o) {
42     auto pool = AsioPool{o, 4};
43     asio::post(pool.executor(), [] {
44         ct::expect(7_i == 0);
45     });
46 };
```

```
C1 [ ===== ] Running 1 test-cases
C2 [ RUN      ] test
C3 Failure in demo.cpp:49
C4 ( 7 == 0 )
C5 [ FAIL     ] test (1.507ms)
C6 [ ===== ] Ran 1 test-cases (1.766ms total)
C7 [ FAIL     ] test
```