# Clean Test

A modern testing framework

Philipp Ochsendorf

November 17th, 2022

## DeepL: We are hiring

DeepL

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

```cpp
#include <clean-test/clean-test.h>

namespace ct = clean_test;
using namespace ct::literals;

auto const s0 = "math"_suite = [] {
  "integer"_test = [] {
    ct::expect(1_i + 2 == 3);
  };
};
```

## Suites and Tests

```cpp
#include <clean-test/clean-test.h>
#include <iostream>

namespace ct = clean_test;
using namespace ct::literals;

auto const talk = "talk"_suite = [] {
  "hello world"_test = [] {
    // Your tests go here
    std::cout << "Hello Meeting C++ 2022!\n";
  };
};
```

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

## Suites and Tests

```cpp
#include <clean-test/clean-test.h>
#include <iostream>

namespace ct = clean_test;
using namespace ct::literals;

auto const talk = "talk"_suite = [] {
  "hello world"_test = [] {
    // Your tests go here
    std::cout << "Hello Meeting C++ 2022!\n";
  };
};

auto const intro = ct::Suite{"intro"} = [] {
  ct::Test{"demo", [] {
    std::cout << "Hi again\n";
  }};
};
```

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

## Suites and Tests

```cpp
#include <clean-test/clean-test.h>
#include <iostream>

namespace ct = clean_test;
using namespace ct::literals;

auto const talk = ”talk”_suite = [] {
  ”hello world”_test = [] {
    // Your tests go here
    std::cout << ”Hello Meeting C++ 2022!\n”;
  };
};

auto const intro = ct::Suite{”intro”} = [] {
  ct::Test{”demo”, [] {
    std::cout << ”Hi again\n”;
  }};

  ”unstable”_tag / ”annoying”_test  = [] {
    std::cout << ”Hi once more.\n”;
  };
};
```

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

## Runtime Configurable

```
> ./demo
```

```
[ ===== ] Running 3 test-cases
[ RUN   ] talk/hello world
[ RUN   ] intro/annoying
Hello Meeting C++ 2022!
Hi once more.
[ RUN   ] intro/demo
Hi again
[ PASS  ] intro/annoying (875.0ns)
[ PASS  ] intro/demo (1.117us)
[ PASS  ] talk/hello world (973.0ns)
[ ===== ] Ran 3 test-cases (1.240ms total)
[ PASS  ] All 3 test-cases
```

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

```
 ├ intro
 │   ├ annoying  {unstable}
 │   └ demo
 └ talk
     └ hello world
```

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

## Expectations

```cpp
#include <clean-test/clean-test.h>

namespace ct = clean_test;
using namespace ct::literals;

auto const t = "expect"_test = [] {
  ct::expect(true);
  ct::expect(0 == 7);
};
```

```
[ ===== ] Running 1 test-cases
[ RUN   ] expect
Failure in demo.cpp:8
false
[ FAIL  ] expect (24.48us)
[ ===== ] Ran 1 test-cases (1.184ms total)
[ FAIL  ] expect
```

## Expectations with Expression Introspection

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

```cpp
#include <clean-test/clean-test.h>

namespace ct = clean_test;
using namespace ct::literals;

auto const t = "expect"_test = [] {
  auto v = 1;
  auto * p = &v;
  ct::expect(nullptr == ct::lift(p));
  ct::expect(v == 7_i);
};
```

```
[ ===== ] Running 1 test-cases
[ RUN   ] expect
Failure in demo.cpp:9
( nullptr == 0x7f1ba3d338ac )
Failure in demo.cpp:10
( 1 == 7 )
[ FAIL  ] expect (36.33us)
[ ===== ] Ran 1 test-cases (1.458ms total)
[ FAIL  ] expect
```

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

## Expression Introspection Attempt (1/4)

```cpp
struct Tag{};

template <typename T>
class Clause final : public Tag {
  T m_value;

public:
  template <typename U>
  constexpr Clause(U && value)
  : m_value{std::forward<U>(value)} {}

  constexpr decltype(auto) value() const {
    return m_value;
  }

  constexpr explicit operator bool() const {
    return static_cast<bool>(value());
  }

  friend std::ostream & operator<<(
    std::ostream & out, Clause const & clause) {
    return out << clause.value();
  }
};
```

```cpp
template <typename T>
concept Expression =
  std::derived_from<std::remove_cvref_t<T>, Tag>;

template <typename T>
constexpr decltype(auto) lift(T && t) {
  if constexpr (Expression<T>) {
    return std::forward<T>(t);
  } else {
    return Clause<T>{std::forward<T>(t)};
  }
}
```

# Expression Introspection Attempt (2/4)

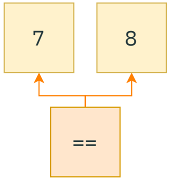
```cpp
template <Expression Lhs, Expression Rhs>
class Equal final : public Tag {
  Lhs m_lhs;
  Rhs m_rhs;

public:
  constexpr Equal(auto && lhs, auto && rhs)
  : m_lhs{std::forward<decltype(lhs)>(lhs)},
    m_rhs{std::forward<decltype(rhs)>(rhs)} {}

  constexpr auto value() const {
    return m_lhs.value() == m_rhs.value();
  }

  constexpr explicit operator bool() const {
    return static_cast<bool>(value());
  }

  friend std::ostream & operator<<(
    std::ostream & out, Equal const & eq) {
    return out << "( " << eq.m_lhs
               << " == " << eq.m_rhs << " )";
  }
};
```

```cpp
template <typename Lhs, typename Rhs>
Equal(Lhs &&, Rhs &&) -> Equal<Lhs, Rhs>;

template <typename Lhs, typename Rhs>
constexpr auto operator==(Lhs && lhs, Rhs && rhs) {
  return Equal{
    lift(std::forward<Lhs>(lhs)),
    lift(std::forward<Rhs>(rhs)),
  };
}
```

## Expression Introspection Attempt (3/4)

```cpp
template <typename T>
void expect(T && t) {
  using Tags = std::array<std::string_view, 2>;
  static auto const tags = Tags{"FAIL", "OK  "};

  std::cout
    << tags[static_cast<bool>(t)]
    << ": " << t << std::endl;
}
int main() {
  static_assert(7 == ct::lift(7));
  ct::expect(7 == ct::lift(8));
}
```

```
C1  FAIL: ( 7 == 8 )
```



## Required Operators

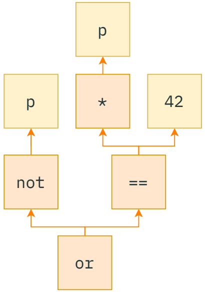
**Overloaded Operators**

| | |
|---|---|
| Unary | `not`, `+`, `-`, `*` (dereference) |
| Arithmetic | `+`, `-`, `*`, `/`, `%` |
| Bitwise | `&`, `\|`, `~`, `^` |
| Comparison | `<`, `<=`, `>`, `>=`, `==`, `!=` |
| Special | `,` (comma), `and`, `or` |

# Expression Introspection Attempt (4/4)

```cpp
template <Expression Lhs, Expression Rhs>
class Or final : public Tag {
  Lhs m_lhs;
  Rhs m_rhs;

public:
  constexpr Or(auto && lhs, auto && rhs)
  : m_lhs{std::forward<decltype(lhs)>(lhs)},
    m_rhs{std::forward<decltype(rhs)>(rhs)} {}

  constexpr decltype(auto) value() const {
    return m_lhs.value() or m_rhs.value();
  }

  constexpr explicit operator bool() const {
    return static_cast<bool>(value());
  }

  friend std::ostream & operator<<(
    std::ostream & out, Or const & o) {
    return out << "( " << o.m_lhs
               << " or " << o.m_rhs << " )";
  }
};
```

```cpp
int main() {
  static_assert(7 == ct::lift(7));
  ct::expect(7 == ct::lift(8));

  int const * const p = nullptr;
  ct::expect(
    not ct::lift(p) or *ct::lift(p) == 42);
}
```

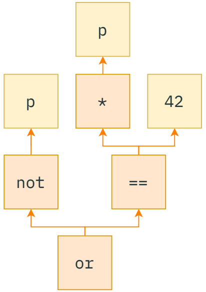


# Expression Introspection Attempt (4/4)

```cpp
template <Expression Lhs, Expression Rhs>
class Or final : public Tag {
  Lhs m_lhs;
  Rhs m_rhs;

public:
  constexpr Or(auto && lhs, auto && rhs)
  : m_lhs{std::forward<decltype(lhs)>(lhs)},
    m_rhs{std::forward<decltype(rhs)>(rhs)} {}

  constexpr decltype(auto) value() const {
    return m_lhs.value() or m_rhs.value();
  }

  constexpr explicit operator bool() const {
    return static_cast<bool>(value());
  }

  friend std::ostream & operator<<(
    std::ostream & out, Or const & o) {
    return out << "( " << o.m_lhs
               << " or " << o.m_rhs << " )";
  }
};
```

```cpp
int main() {
  static_assert(7 == ct::lift(7));
  ct::expect(7 == ct::lift(8));

  int const * const p = nullptr;
  ct::expect(
    not ct::lift(p) or *ct::lift(p) == 42);
}
```

```
FAIL: ( 7 == 8 )
segmentation fault (core dumped)
```



## Expression Introspection Cached Attempt (1/2)

```
template <typename Expression>
using Value =
  decltype(std::declval<Expression>().value());

template <Expression Lhs, Expression Rhs>
class CachingOr final : public Tag {
  Lhs m_lhs;
  Rhs m_rhs;
  Value<Lhs> m_lhs_value;
  std::optional<Value<Rhs>> m_rhs_value;

public:
  constexpr CachingOr(auto && lhs, auto && rhs)
  : m_lhs{std::forward<decltype(lhs)>(lhs)},
    m_rhs{std::forward<decltype(rhs)>(rhs)},
    m_lhs_value{m_lhs.value()},
    m_rhs_value{m_lhs_value
      ? std::nullopt
      : std::make_optional(m_rhs.value())}
  {}

  constexpr decltype(auto) value() const {
    return m_lhs_value or *m_rhs_value;
  }

  constexpr explicit operator bool() const {
    return static_cast<bool>(value());
  }

  friend std::ostream & operator<<(
    std::ostream & out, CachingOr const & o)
  {
    out << "( " << o.m_lhs << " or ";
    if (o.m_rhs_value) {
      out << o.m_rhs;
    } else {
      out << "<unknown>";
    }
    return out << " )";
  }
};
```

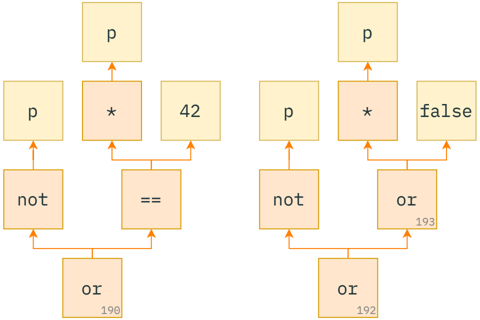
## Expression Introspection Cached Attempt (2/2)

```
int main() {
  static_assert(7 == ct::lift(7));
  ct::expect(7 == ct::lift(8));

  int const * const p = nullptr;
  ct::expect(
    not ct::lift(p) or *ct::lift(p) == 42);
}
```

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

# Expression Introspection Cached Attempt (2/2)

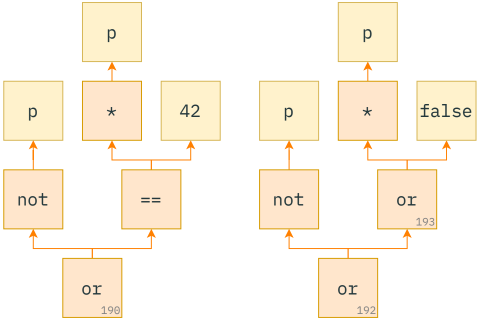
```
184 int main() {
185   static_assert(7 == ct::lift(7));
186   ct::expect(7 == ct::lift(8));
187
188   int const * const p = nullptr;
189   ct::expect(
190     not ct::lift(p) or *ct::lift(p) == 42);
191   ct::expect(
192     not ct::lift(p) or
193     (*ct::lift(p) or false));
194 }
```



```
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 }
```



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

```cpp
template <Expression Lhs, Expression Rhs>
class Or final : public Tag {
  Lhs m_lhs;
  Rhs m_rhs;

public:
  using Evaluation = OrEvaluation<typename Lhs::Evaluation, typename Rhs::Evaluation>;
  friend Evaluation;

  constexpr Or(auto && lhs, auto && rhs)
  : m_lhs{std::forward<decltype(lhs)>(lhs)},
    m_rhs{std::forward<decltype(rhs)>(rhs)}
  {}

  [[nodiscard]] constexpr auto evaluation() const {
    return Evaluation{*this};
  }

  constexpr explicit operator bool() const {
    return static_cast<bool>(evaluation().value());
  }
};
```

## Expression Evaluation with Introspection (2/3)

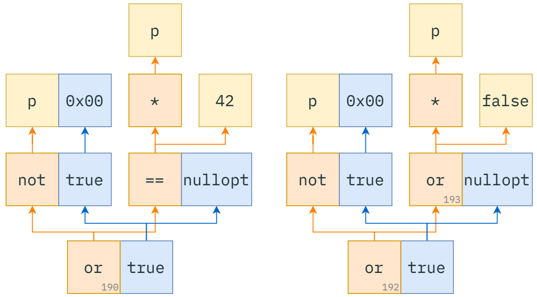
```cpp
template <typename Lhs, typename Rhs>
class OrEvaluation final {
  Lhs m_lhs;
  std::optional<Rhs> m_rhs;
  bool m_value;

public:
  OrEvaluation(auto && expr)
  : m_lhs{expr.m_lhs.evaluation()},
    m_rhs{
      static_cast<bool>(m_lhs.value())
      ? std::nullopt
      : std::optional{
          expr.m_rhs.evaluation()
        }
    },
    m_value{
      m_rhs
      ? static_cast<bool>(m_rhs->value())
      : static_cast<bool>(m_lhs.value())
    }
  {}

  constexpr decltype(auto) value() const {
    return m_value;
  }

  friend std::ostream & operator<<(
    std::ostream & out, OrEvaluation const & eval) {
    out << "( " << eval.m_lhs << " or ";
    if (eval.m_rhs) {
      out << *eval.m_rhs;
    } else {
      out << "<unknown>";
    }
    return out << " )";
  }
};
```

## 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

```cpp
int * p = nullptr;
ct::expect(not ct::lift(p) or *ct::lift(p)); // ok

ct::expect(not ct::lift(p) or use(*p)); // ERROR
// Alternative 1
if (p) {
  ct::expect(use(*p));
}
// Alternative 2: lifting callable
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 }));
```

```
[ ===== ] Running 5 test-cases
[ RUN   ] advanced/debug output
Failure in demo.cpp:22
false
but why?
[ FAIL  ] advanced/debug output (12.70us)
[ RUN   ] advanced/asserted
Failure (asserted) in demo.cpp:26
false
[ ABORT ] advanced/asserted (24.72us)
[ RUN   ] advanced/flaky
Failure (flaky) in demo.cpp:33
false
[ PASS  ] advanced/flaky (4.243us)
[ RUN   ] advanced/throw
[ PASS  ] advanced/throw (12.59us)
[ RUN   ] advanced/abort
[ PASS  ] advanced/abort (79.36ms)
[ ===== ] Ran 5 test-cases (79.72ms total)
[ FAIL  ] advanced/debug output
[ FAIL  ] advanced/asserted
[ PASS  ] All other 3 test-cases
```

## Advanced Distance Checks

```cpp
static_assert(ct::utils::norm(-2) == 2_i);

auto const x = 0.15_d + 0.15;
auto const y = 0.1_d + 0.2;

ct::expect(x == y);
ct::expect(ct::is_close(x, y));
ct::expect(ct::is_close(x, y + .5));

ct::expect(
  ct::distance(x, y + .5) <= ct::tolerance(.5));
```

- CPO `ct::utils::norm`
- Absolute distance

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

- Relative distance

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

```
[ ===== ] Running 1 test-cases
[ RUN   ] distance/close
Failure in demo.cpp:22
( ( 0.15 + 0.15 ) == ( 0.1 + 0.2 ) )
Failure in demo.cpp:24
( distance(( 0.15 + 0.15 ), ( ( 0.1 + 0.2 ) + 0.5 )) = {absolute: 0.5, relative: 0.625}
 <= {absolute: 2.22045e-16, relative: 2.22045e-16} )
[ FAIL  ] distance/close (88.75us)
[ ===== ] Ran 1 test-cases (246.5us total)
[ FAIL  ] distance/close
```

# Conversion Tools

clean-test/migration

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

*Original*

```cpp
#define BOOST_TEST_MAIN
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_SUITE(demo)

BOOST_AUTO_TEST_CASE(talk)
{
  BOOST_WARN(2 + 2 == 2 * 2);
  BOOST_CHECK_MESSAGE(
    3 + 3 == 2 * 3, "can't touch this");
  BOOST_REQUIRE_EQUAL(4 + 4, 2 * 4);
}

BOOST_AUTO_TEST_SUITE_END()
```

*Converted*

```cpp
#include <clean-test/clean-test.h>

namespace ct = clean_test;
using namespace ct::literals;

auto const demo = "demo"_suite = [] {

"talk"_test = []
{
  ct::expect(2_i + 2 == 2_i * 2) << ct::flaky;
  ct::expect(
    3_i + 3 == 2_i * 3) << "can't touch this";
  ct::expect(4_i + 4 == 2_i * 4) << ct::asserted;
};

};
```

# 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

```cpp
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 };
```

```
[ ===== ] Running 1 test-cases
[ RUN   ] par
[ PASS  ] par (753.6us)
[ ----- ] Warning: Observed test-expectations at unknown Observer.
[       ]   - demo.cpp:27
[ ----- ] This is likely caused by missing to propagate an Observer.
[ ===== ] Ran 1 test-cases (1.076ms total)
[ PASS  ] All 1 test-cases
```

## Data Driven Tests

```cpp
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 };
```

```
[ ===== ] Running 4 test-cases
[ RUN   ] data/static/0
Failure in demo.cpp:24
( 0 > 0 )
[ FAIL  ] data/static/0 (14.06us)
[ RUN   ] data/static/1337
[ PASS  ] data/static/1337 (5.829us)
[ RUN   ] data/temporary/0
[ PASS  ] data/temporary/0 (10.19us)
[ RUN   ] data/temporary/1337
Failure in demo.cpp:29
( "1337" != "1337" )
[ FAIL  ] data/temporary/1337 (10.42us)
[ ===== ] Ran 4 test-cases (195.6us total)
[ FAIL  ] data/static/0
[ FAIL  ] data/temporary/1337
[ 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

```cpp
class AsioPool {
  asio::io_context m_context;
  std::optional<asio::io_context::work> m_work;
  std::vector<std::jthread> m_workers;

public:
  AsioPool(
    ct::Observer & o,
    std::size_t const n)
  : m_context{},
    m_work{m_context}
  {
    while(m_workers.size() < n) {
      m_workers.emplace_back([&, this] {
        auto const os = ct::ObservationSetup{o};
        m_context.run();
      });
    }
  }

  ~AsioPool() {
    m_work.reset();
  }
```

```cpp
  auto executor() {
    return m_context.get_executor();
  }
};


auto test = "test"_test = [](ct::Observer & o) {
  auto pool = AsioPool{o, 4};
  asio::post(pool.executor(), [] {
    ct::expect(7_i == 0);
  });
};
```

```
[ ===== ] Running 1 test-cases
[ RUN   ] test
Failure in demo.cpp:49
( 7 == 0 )
[ FAIL  ] test (1.142ms)
[ ===== ] Ran 1 test-cases (1.443ms total)
[ FAIL  ] test
```