TOOLS AND TECHNIQUES FOR TESTING CALLBACKS

TO KILL A MOCKING FRAMEWORK

CALLBACKS ARE EVERYWHERE

- Asynchronous I/O Events
- Observer Pattern
- Customization/extension points
- Functional Programming

```
auto square_add = [] (auto a, auto b) {
  return a * a + b;
};
int is[] = { 1, 2, 3 };
auto q = std::accumulate(std::begin(is), std::end(is), 0, square_add);
```

TOKENIZER (VERSION 1)

```
template <typename callback_t>
struct tokenizer1 {
  callback_t& callback_;
  tokenizer1(callback_t& callback) : callback_{callback} {}

  void operator()(std::string_view input) const {
    E char ws[] = " \t\n";
    size_t a{}, b{};
    do {
      a = input.find_first_not_of(ws, b);
      if (a == input.npos) return;
      b = input.find_first_of(ws, a);
      auto tok = input.substr(a, b == input.npos? b : b - a);
      // call callback here!
      callback_.string_token(tok);
    } while (b != input.npos);
  }
};
```

TESTING TOKENIZER

```
/* what type goes here?? */ mock_callback;
tokenizer1 test_me{mock_callback};
test_me("hello");
/* what verification code goes here? */
```

- Verify the callback is called correct number of times...
- ... with the correct parameters
- ... and returning the correct values to the code under test (if any)

GOOGLE MOCK

```
struct tokenizer1_test : ::testing::Test {
  struct mock_callback {
    MOCK_METHOD1(string_token, void(std::string_view));
  };
  mock_callback callback_;
  tokenizer1<mock_callback> tokenizer_{callback_};
};

TEST_F(tokenizer1_test, hello) {
  EXPECT_CALL(callback_, string_token("hello"sv)).Times(1);
  tokenizer_("hello");
}
```

GOOGLE MOCK “MATCHERS”

```
EXPECT_CALL(foo, Blah(_, _, _))
      .With(AllOf(Args<0, 1>(Lt()), Args<1, 2>(Lt())));

EXPECT_CALL(foo, InRange(Ne(0), _))
      .With(Lt());

EXPECT_CALL(foo, DoThis(Lt(5)))
      .WillRepeatedly(Return('a'));
```

MOCKING WITH STD::FUNCTION

```
struct tokenizer1_fixture {
  struct mock_callback {
    std::function<void(std::string_view)> string_token;
  };
  mock_callback callback_;
  tokenizer1<mock_callback> tokenizer_{callback_};
};

BOOST_FIXTURE_TEST_CASE(hello_expect_calls, tokenizer1_fixture) {
  callback_.string_token = expect_calls(1, [] (std::string_view seen) {
      BOOST_TEST(seen == "hello");
    });
  tokenizer_("hello");
}
```

DECORATED CALLABLE

```
template <typename decorator_t, typename callable_t>
struct decorated_callable {
  decorator_t decorator_;
  callable_t callable_;

  template <typename ... args_t>
  auto operator() (args_t&& ... args) {
    decorator_(std::forward<args_t>(args)...);
    return callable_(std::forward<args_t>(args)...);
  }
};
```

CALL COUNT CHECKER (VERSION 0)

```
// this code is flawed - for exposition only
struct call_count_checker {
  int calls_;
  int expected_;
  template <typename ... args_t>
  void operator() (args_t&&...) {
    ++calls_;
  }
  ~call_count_checker() {
    BOOST_TEST(calls_ == expected_);
  }
};

template <typename Fn>
auto expect_calls(int expected, Fn fn) {
  return decorated_callable<call_count_checker, Fn>{{0, expected}, fn};
}
```

CALL COUNT CHECKER (VERSION 1)

```
using source_location = std::experimental::source_location;

// nested class within call_count_checker
struct state {
  state(source_location&& location, unsigned expected)
    : location_{std::move(location)}, expected_{expected}
  {}
  ~state() {
    BOOST_TEST(expected_ == calls_,
      "Function defined at " << location_.file_name() << ':' << location_.line()
      << " expected " << *expected_ << " calls, " << calls_ << " seen");
    }
  }
  const source_location location_;
  const unsigned expected_;
  unsigned calls_ {};
};
```

```
class call_count_checker {
  struct state { /* ... */  };
  std::shared_ptr<state> state_;
public:
  call_count_checker(unsigned expected,
                     source_location&& location = source_location::current())
    : state_{std::make_shared<state>(std::move(location), expected)}
  {}
  call_count_checker(const call_count_checker&) = default;
  call_count_checker(call_count_checker&&) = default;

  template <typename ... Args>
  void operator() (Args&& ...) const {
    state_->calls_ += 1;
  }
};
```

EXPECT_CALLS FAILURE

```
BOOST_AUTO_TEST_CASE(hello_expect_calls_fail, *boost::unit_test::expected_failures(1))
{
  callback_.string_token = expect_calls(1, [] (std::string_view) {});
  tokenizer_("");
}
```

```
../mocking.hpp(75): error: in "tokenizer1_test/hello_expect_calls_fail":
Function defined at ../tokenizer1_test2.cpp:43 expected 1 calls, 0 seen
```

VARYING PARAMETERS

```
BOOST_AUTO_TEST_CASE(hello_world_1) {
  auto expect_world = expect_calls(1, [&] (std::string_view seen) {
      BOOST_TEST(seen == "world");
    });
  auto expect_hello = expect_calls(1, [&] (std::string_view seen) {
      BOOST_TEST(seen == "hello");
      callback_.string_token = expect_world;
    });
  callback_.string_token = expect_hello;
  tokenizer_("hello world");
}
```

```
BOOST_AUTO_TEST_CASE(hello_world_2) {
  const char* exp_toks[] = { "hello", "world" };
  callback_.string_token = expect_calls(2,
    [exp=exp_toks] (std::string_view seen) mutable {
      BOOST_TEST(seen == *exp++);
    });
  tokenizer_("hello world");
}
```

NON-DETERMINISTIC ORDERING

```
BOOST_AUTO_TEST_CASE(hello_world) {
  std::set expected { "hello"sv, "world"sv };
  callback_.string_token = expect_calls(2, [&] (std::string_view tok) {
      BOOST_TEST(expected.erase(tok) == 1, word << " not expected");
    });
  tokenizer_("hello world");
}
```

TOKENIZER (VERSION 2)

```
template <typename callback_t>
struct tokenizer2 {
  callback_t& callback_;
  tokenizer2(callback_t& c) : callback_{c} {}

  void operator()(std::string_view input) const {
    const char ws[] = " \t\n";
    size_t a{}, b{};
    do {
      a = input.find_first_not_of(ws, b);
      if (a == input.npos) return;
      b = input.find_first_of(ws, a);
      auto tok = input.substr(a, b == input.npos? b : b - a);
      if (int i; std::from_chars(tok.begin(), tok.end(), i).ec == std::errc{}) {
        callback_.int_token(i);
      } else {
        callback_.string_token(tok);
      }
    } while (b != input.npos);
  }
};
```

ORDERED CALLBACKS

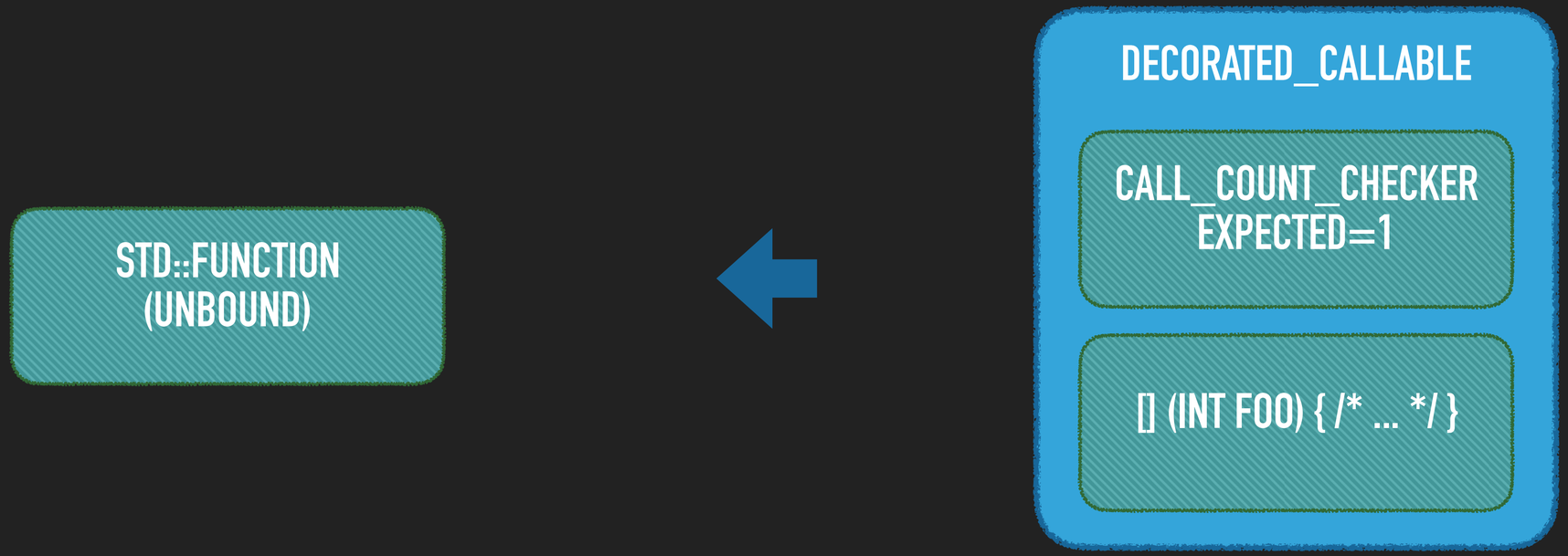
```
BOOST_AUTO_TEST_CASE(hello_123_variant) {
  std::deque<std::variant<std::string_view, int>> exp {{ "hello"sv, 123 }};

  auto check_token = expect_calls(size(exp), [&] (auto seen) {
    auto* next_exp = std::get_if<decltype(seen)>(&exp.front());
    BOOST_REQUIRE(next_exp);
    BOOST_TEST(*next_exp == seen);
    exp.pop_front();
    });
  callback_.int_token = check_token;
  callback_.string_token = check_token;
  tokenizer_("hello 123");
}
```

EXPOSING CALL COUNT CHECKER

```
BOOST_AUTO_TEST_CASE(hello_123) {
  callback_.string_token = expect_calls(1, [] (std::string_view seen) {
      BOOST_TEST(seen == "hello");
    });
  callback_.int_token = expect_calls(1, [] (int seen) {
      // want to check calls to string_token == 1 here
      BOOST_TEST(seen == 123);
    });
  tokenizer_("hello 123");
}
```

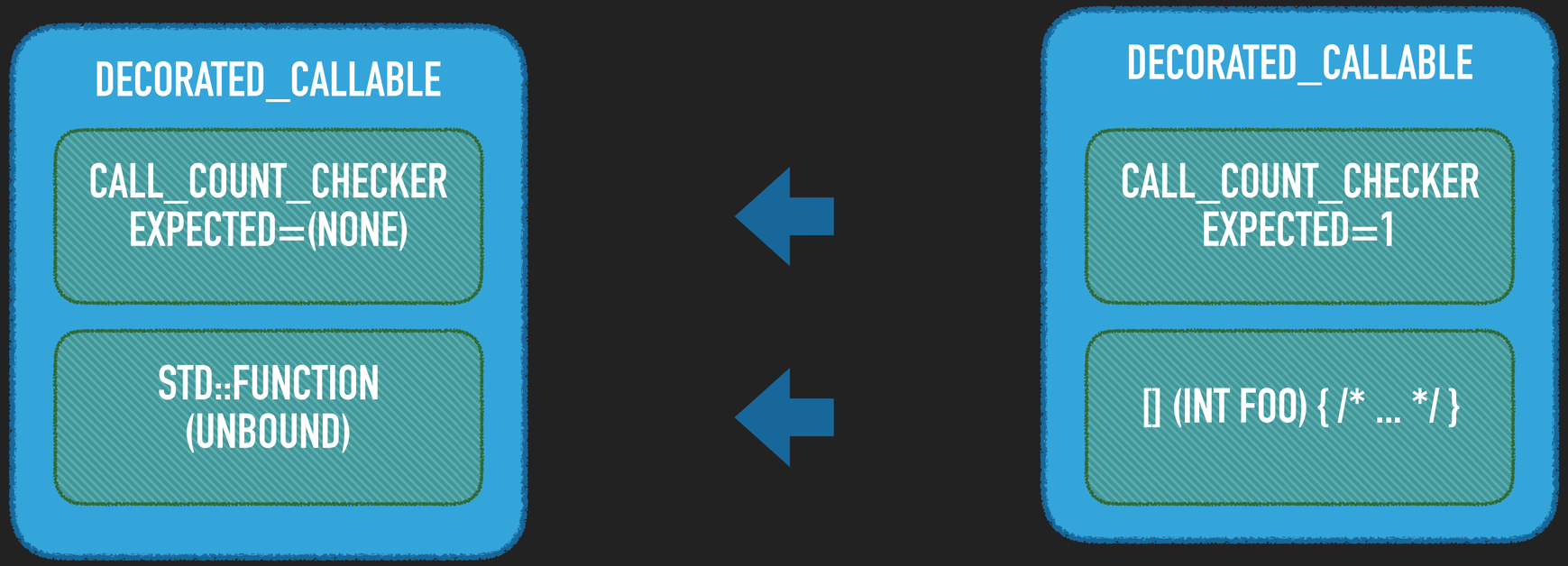
RE-DECORATING

```
a_mock_function = expect_calls(1, [] (int foo) { /* ... */ });
```



RE-DECORATING

```
a_mock_function = expect_calls(1, [] (int foo) { /* ... */ });
```



COUNTED_FUNCTION

```
template <typename>
struct counted_function;

template <typename Ret, typename ... Args>
struct counted_function<Ret(Args...)>
  : decorated_callable<call_count_checker, std::function<Ret(Args...)>>
{
  using base_t = decorated_callable<call_count_checker, std::function<Ret(Args...)>>;

  using base_t::decorated_callable;
  using base_t::operator=;
  using base_t::decorator;

  auto current_count() const { return decorator().current_count(); }
};
```

PUTTING IT ALL TOGETHER

```
struct tokenizer2_fixture {
  struct mock_callback {
    counted_function<void(std::string_view)> string_token;
    counted_function<void(int)> int_token;
  };

  mock_callback callback_;
  tokenizer2<mock_callback> tokenizer_{callback_};
};
```

```
BOOST_AUTO_TEST_CASE(hello_123_ordered) {
  callback_.string_token = expect_calls(1, [] (std::string_view seen) {
      BOOST_TEST(seen == "hello");
    });
  callback_.int_token = expect_calls(1, [&] (int seen) {
      BOOST_TEST(callback_.string_token.current_count() == 1);
      BOOST_TEST(seen == 123);
    });
  tokenizer_("hello 123");
}
```

LIMITATION: OVERLOADING

```
struct mock_callback {
  std::function<void(std::string_view)> token_string_view_;
  std::function<void(int)> token_int_;

  void token(std::string_view s) { token_string_view_(s); }
  void token(int i) { token_int_(i); }
};
```

LIMITATION: VIRTUAL FUNCTIONS

```
struct callback_interface {
  virtual void string_token(std::string_view) = 0;
};
struct mock_callback : callback_interface {
  std::function<void(std::string_view)> string_token_fn_;
  void string_token(std::string_view arg) override { string_token_fn_(arg); }
};
```

LIMITATION: DUPLICATED FUNCTION SIGNATURES

```
struct {
   std::function<void(std::string_view, int, double) foo;
} callback;

callback.foo = [&] (std::string_view s, int i, double d) { /* */ };
```

TAKEAWAYS

- Consider whether a declarative or imperative syntax is better for your tests
- `std::function` can be used to build mock objects...
- ... but don't forget to ensure that mocked functions are called somehow!
- Standard containers are useful for representing sequences of expectations
- Decorators can be used to augment test code

WHERE TO FIND ME

- github.com/randomphrase/mocking - source code for this talk
- @randomphrase - twitter
- girtby.net - blog archives
- alastair@girtby.net - email
- @randomphr4se - instagram, mainly pictures of cats and cocktails