Using Trompeloeil
a mocking framework for modern C++

Björn Fahller

Using Trompeloeil
a mocking framework for modern C++

Trompe-l'œil noun (Concise Encyclopedia)

Style of representation in which a painted object is intended to deceive the viewer into believing it is the object itself...

Björn Fahller

Using Trompeloeil
a mocking framework for modern C++

Trompe-l'œil noun (Concise Encyclopedia)

Style of representation in which a painted object is intended to deceive the viewer into believing it is the object itself...

Björn Fahller

Using Trompeloeil
a mocking framework for modern C++

Trompe-l'œil noun (Concise Encyclopedia)

Style of representation in which a painted object is intended to deceive the viewer into believing it is the object itself...

Björn Fahller

Trompeloeil is:

Pure C++14 without any dependencies

Implemented in a single header file

Under Boost Software License 1.0

Available from Conan 

Adaptable to any (that I know of) unit testing framework

https://github.com/rollbear/trompeloeil

Documentation

- **Integrating with unit test frame works**
- Intro presentation from Stockholm C++ UG (YouTube)
- Introduction
- Trompeloeil on CppCast
- Cheat Sheet (2*A4)
- Cook Book
- FAQ
- Reference

Integrating with unit test frame works

By default, *Trompeloeil* reports violations by throwing an exception, explaining the problem in the `what()` string.

Depending on your test frame work and your runtime environment, this may, or may not, suffice.

Trompeloeil offers support for adaptation to any test frame work. Some sample adaptations are:

- Catch!
- crpcut
- gtest
- ...

Integrating with unit test frame works

By default, *Trompeloeil* reports violations by throwing an exception, explaining the problem in the `what()` string.

Depending on your test frame work and your runtime environment, this may, or may not, suffice.

Trompeloeil offers support for adaptation to any test frame work. Some sample adaptations are:

- Catch!
- crpcut
- gtest
- ...

If your favourite unit testing frame work is not listed, please write an adapter for it, document it in the CookBook and submit a pull request.

Introduction by example.

Free improvisation around the theme in Martin Fowler's whisky store order example, from the blog post "Mocks Aren't Stubs"

`http://martinfowler.com/articles/mocksArentStubs.html`

```
class order {
  ...
};
```

This is the class to implement.

Introduction by example.

Free improvisation around the theme in Martin Fowler's whisky store order example, from the blog post "Mocks Aren't Stubs"

`http://martinfowler.com/articles/mocksArentStubs.html`

```
class order {
  ...
};
```


It will communicate with a store

```
class store {
  ...
};
```

Introduction by example.

Free improvisation around the theme in Martin Fowler's whisky store order example, from the blog post "Mocks Aren't Stubs"

`http://martinfowler.com/articles/mocksArentStubs.html`

```
class order {
  ...
};
```



```
class store {
  ...
};
```

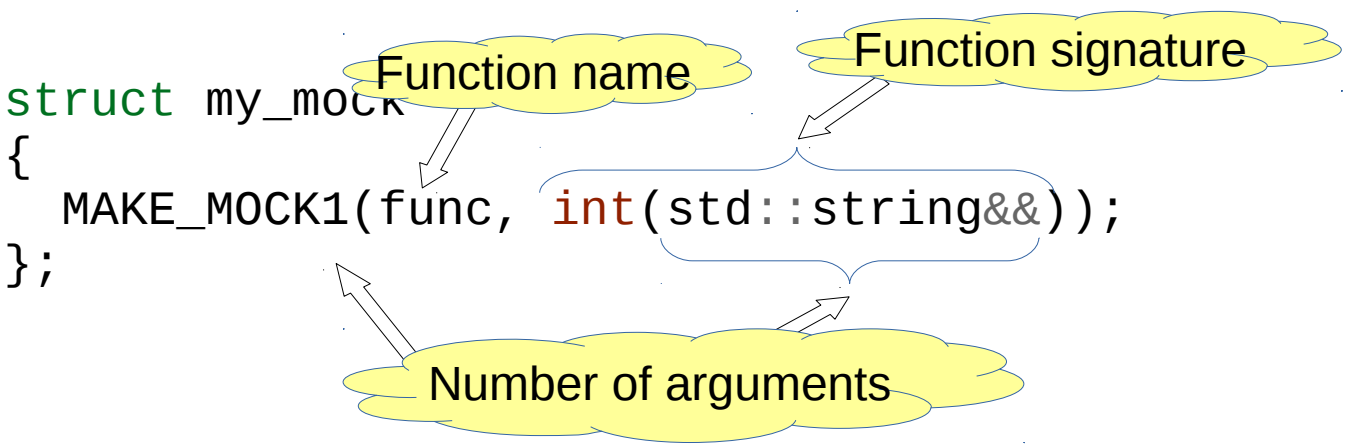
It will communicate with a store.
The store will be mocked.

Creating a mock type.

```
#include <trompeloeil.hpp>




struct my_mock
{
  MAKE_MOCK1(func, int(std::string&&));
};
```

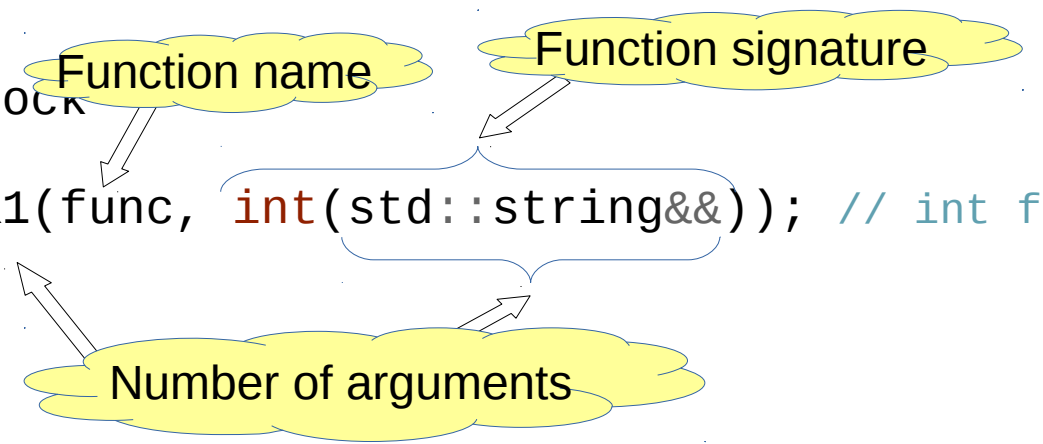
Creating a mock type.

```
#include <trompeloeil.hpp>
```



Creating a mock type.

```
#include <trompeloeil.hpp>
```




```
struct my_mock
{
  MAKE_MOCK1(func, int(std::string&&)); // int func(std::string&&);
};
```

Creating a mock type.

```
#include <trompeloeil.hpp>
```

```
struct my_mock
{
  MAKE_MOCK2(func, int(std::string&&)); // int func(std::string&&);
};
```

```
In file included from cardinality_mismatch.cpp:1:0:
trompeloeil.hpp:2953:3: error: static assertion failed: Function signature does not have 2
parameters
  static_assert(TROMPELOEIL_ID(cardinality_match)::value,                          \
  ^
trompeloeil.hpp:2885:3: note: in expansion of macro ˜TROMPELOEIL_MAKE_MOCK_′
  TROMPELOEIL_MAKE_MOCK_(name,,2, __VA_ARGS__,,)
  ^
trompeloeil.hpp:3209:35: note: in expansion of macro ˜TROMPELOEIL_MAKE_MOCK2′
#define MAKE_MOCK2                TROMPELOEIL_MAKE_MOCK2
                                  ^
cardinality_mismatch.cpp:4:3: note: in expansion of macro ˜MAKE_MOCK2′
  MAKE_MOCK2(func, int(std::string&&));
  ^
```

```
struct my_mock
{
  MAKE_MOCK2(func, int(std::string&&)); // int func(std::string&&);
};
```

```
In file included from cardinality_mismatch.cpp:1:0:
trompeloeil.hpp:2953:3: error: static assertion failed: Function signature does not have 2
parameters
  static_assert(TROMPELOEIL_ID(cardinality_match)::value,                          \
  ^
trompeloeil.hpp:2885:3: note: in expansion of macro ‘TROMPELOEIL_MAKE_MOCK_’
  TROMPELOEIL_MAKE_MOCK_(name,,2, __VA_ARGS__,,)
  ^
trompeloeil.hpp:3209:35: note: in expansion of macro ‘TROMPELOEIL_MAKE_MOCK2’
#define MAKE_MOCK2                TROMPELOEIL_MAKE_MOCK2
                                  ^
cardinality_mismatch.cpp:4:3: note: in expansion of macro ‘MAKE_MOCK2’
  MAKE_MOCK2(func, int(std::string&&));
  ^
```

Full error message from g++ 5.4

```
struct my_mock
{
  MAKE_MOCK2(func, int(std::string&&)); // int func(std::string&&);
};
```

Oh no, horrible mistake!

```
#include <trompeloeil.hpp>
```

```
struct my_mock
{
  MAKE_MOCK1(func, int(std::string&&)); // int func(std::string&&);
};
```

Creating a mock type.

```
#include <trompeloeil.hpp>

struct interface
{
  virtual ~interface() = default;
  virtual int func(std::string&&) = 0;
};

struct my_mock : public interface
{
  MAKE_MOCK1(func, int(std::string&&));
};
```

Creating a mock type.

```
#include <trompeloeil.hpp>

struct interface
{
  virtual ~interface() = default;
  virtual int func(std::string&&) = 0;
};

struct my_mock : public interface
{
  MAKE_MOCK1(func, int(std::string&&), override);
};
```

Improvisation around **http://martinfowler.com/articles/mocksArentStubs.html**

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};
```

Improvisation around **http://martinfowler.com/articles/mocksArentStubs.html**

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

class order {
public:
  void add(const std::string article, size_t quantity);
  void fill(store&);
};
```

```
order the_order;
the_order.add("Talisker", 50);
store& a_store = …
the_order.fill(a_store);
```

```
class store {
public:
  virtual void add(const std::string article, size_t quantity) = 0;
  virtual bool has_inventory(const std::string article) const = 0;
  virtual void remove(const std::string article, size_t quantity) = 0;
};

class warehouse : public store {
public:
  void add(const std::string article, size_t quantity);
  void fill(store&);
};
```

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);

};
```

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))



    test_order.fill(store);
  }
}
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;                                    Create an order object
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))



    test_order.fill(store);
  }
}
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))



    test_order.fill(store);
  }
}
```

Save whiskies to order – no action

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))



    test_order.fill(store);
  }
}
```

Create the mocked store – no action

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))

    test_order.fill(store);
  }
}
```

Set up expectation for call

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))

    test_order.fill(store);
  }
}
```

Set up expectation for call

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));


    test_order.fill(store);
  }
}
```

Can call `store.inventory(whisky)`
Can compare `const char*` *and* `const std::string&`

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));


    test_order.fill(store);
  }
}
```

Creates an “anonymous” expectation object

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));


    test_order.fill(store);
  }
}
```

Parameters are copied into the expectation object.

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));


    test_order.fill(store);
  }
}
```

Adds entry first in expectation list for `inventory(const std::string)`

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));


    test_order.fill(store);
  }
}
```

Expectation must be fulfilled before destruction of the expectation object at the end of scope

Test by setting up expectations

```
class mock_store : public store {
public:
  MAK                                                    d::string&), override);
  MAK                                                     size_t), override);
};

TEST_                                                    nsufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));



    test_order.fill(store);
  }
}
```

```
class order
{
public:
  void add(const std::string&, size_t) {}
  void fill(store&) {}
};
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAK
  MAK
};

TEST_
{
  orc
  tes
  moc
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky));



    test_order.fill(store);
  }
}
```

```
In file included from order_test.cpp:1:0:
/home/bjorn/devel/trompeloeil/trompeloeil.hpp: In instantiation of 'auto
trompeloeil::call_validator_t<Mock>::operator+(trompeloeil::call_modifier<M,
Tag, Info>&&) const [with M = trompeloeil::call_matcher<long unsigned int(const
std::basic_string<char>&), std::tuple<const char*> >; Tag =
mock_store::trompeloeil_tag_type_trompeloeil_7; Info =
trompeloeil::matcher_info<long unsigned int(const std::basic_string<char>&)>;
Mock = mock_store]':
order_test.cpp:23:5:   required from here
/home/bjorn/devel/trompeloeil/trompeloeil.hpp:3155:7: error: static assertion
failed: RETURN missing for non-void function
       static_assert(valid_return_type, "RETURN missing for non-void function");
       ^~~~~~~~~~~~~
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAK
  MAK
};

TEST_
{
  ord
  tes
  moc
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))


    test_order.fill(store);
  }
}
```

```
In file included from order_test.cpp:1:0:
/home/bjorn/devel/trompeloeil/trompeloeil.hpp: In instantiation of 'auto
trompeloeil::call_validator_t<Mock>::operator+(trompeloeil::call_modifier<M,
Tag, Info>&&) const [with M = trompeloeil::call_matcher<long unsigned int(const
std::basic_string<char>&), std::tuple<const char*> >; Tag =
mock_store::trompeloeil_tag_type_trompeloeil_7; Info =
trompeloeil::matcher_info<long unsigned int(const std::basic_string<char>&)>;
Mock = mock_store]':
order_test.cpp:23:5:   required from here
/home/bjorn/devel/trompeloeil/trompeloeil.hpp:3155:7: error: static assertion
failed: RETURN missing for non-void function
       static_assert(valid_return_type, "RETURN missing for non-void function");
       ^~~~~~~~~~~~~
```

Full error message from g++ 6.2

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))
      .RETURN(50);


    test_order.fill(store);
  }
}
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};

TEST_CASE("filling does nothing if stock is insufficient")
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))
      .RETURN(50);


    test_order.fill(store);
  }
}
```

Any expression with a type convertible to the return type of the function.

Test by setting up expectations

```
cla
pub
  M
  M
};

TES
{
  o
  t
  m
  {


  }
}
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a.out is a Catch v1.8.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
fill does nothing if stock is insufficient
-------------------------------------------------------------------------------
order_test.cpp:17
...............................................................................

order_test.cpp:50: FAILED:
  CHECK( failure.empty() )
with expansion:
  false
with message:
  failure := "order_test.cpp:23
  Unfulfilled expectation:
  Expected store.inventory(whisky) to be called once, actually never called
    param  _1 == Talisker
  "

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAK                                                    override);
  MAK                                                    erride);
};

TEST_                                                            )
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))
      .RETURN(50);


    test_order.fill(store);
  }
}
```

```
class order
{
public:
  void add(const std::string& name, size_t) { article = name; }
  void fill(store& the_store) { the_store.inventory(article); }
private:
  std::string article;
};
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAK                                                    override);
  MAK                                                    erride);
};

TEST_                                                          )
{
  order test_order;
  test_order.add("Talisker", 51);
  mock_store store;
  {
    const char* whisky = "Talisker";
    REQUIRE_CALL(store, inventory(whisky))
      .RETURN(50);
```

```
class order
{
public:
  void add(const std::string& name, size_t) { article = name; }
  void fill(store& the_store) { the_store.inventory(article); }
private:
  std::string article;
};
```

```
===============================================================================
test cases: 1 | 1 passed
assertions: - none -
```

```
  }
}
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};
TEST_CASE("filling does nothing if stock is insufficient")...
TEST_CASE("filling removes from store if in stock")
{
  order test_order;
  test_order.add("Talisker", 50);
  mock_store store;
  {
    REQUIRE_CALL(store, inventory("Talisker"))
      .RETURN(50);
    REQUIRE_CALL(store, remove("Talisker", 50));

    test_order.fill(store);
  }
}
```

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};
TEST_CASE("filling removes from store if in stock")
{
  order test_order;
  test_order.add("Talisker", 50);
  mock_store store;
  {

    REQUIRE_CALL(store, inventory("Talisker"))
      .RETURN(50);

    REQUIRE_CALL(store, remove("Talisker", 50));

    test_order.fill(store);
  }
}
```

Adds entry to expectation list for `inventory(const std::string&)`

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, size_t), override);
};
TEST_CASE("filling removes from store if in stock")
{
  order test_order;
  test_order.add("Talisker", 50);
  mock_store store;
  {

    REQUIRE_CALL(store, inventory("Talisker"))
      .RETURN(50);

    REQUIRE_CALL(store, remove("Talisker", 50));

    test_order.fill(store);
  }
}
```

Adds entry to expectation list for `inventory(const std::string&)`

Adds entry to expectation list for `remove(const std::string&,size_t)`

Test by setting up expectati

```
class mock
public:
  MAKE_C                                    t std::string&), override);
  MAKE_M                                    ng&, size_t), override);
};
TEST_CASE                                   e if in stock")
{
  order test_o
  test_order.add("Talis                     , 50);
  mock_store store;
  {
    REQUIRE_CALL(store, inventory("Talisker"))
      .RETURN(50);

    REQUIRE_CALL(store, remove("Talisker", 50));

    test_order.fill(store);
  }
}
```

Note that the expectations are added to separate lists.

There is no ordering relation between them, so there are two equally acceptable sequences here.

Adds entry to expectation list for `inventory(const std::string&)`

Adds entry to expectation list for `remove(const std::string&,size_t)`

Test by setting up expectations

```
class mock_store : public store {
public:
  MAKE_CONST_MOCK1(inventory, size_t(const std::string&), override);
  MAKE_MOCK2(remove, void(const std::string&, unsigned), override);
};
TEST_CASE("filling removes from store inventory")
{
  order test_order;
  test_order.add("Talisker", 50);
  mock_store store;
  {
    trompeloeil::sequence seq;
    REQUIRE_CALL(store, inventory("Talisker"))
      .RETURN(50)
      .IN_SEQUENCE(seq);
    REQUIRE_CALL(store, remove("Talisker", 50))
      .IN_SEQUENCE(seq);
    test_order.fill(store);
  }
}
```

Sequence objects provides a way to impose and enforce a sequential ordering of otherwise unrelated expectations.

Test

```
cla
pub
  M
  M
};
TES
{
  o
  t
  m
  {
  }
}
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a.out is a Catch v1.8.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
filling removes from store if in stock
-------------------------------------------------------------------------------
order_test.cpp:31
...............................................................................

order_test.cpp:64: FAILED:
  CHECK( failure.empty() )
with expansion:
  false
with message:
  failure := "order_test.cpp:39
  Unfulfilled expectation:
  Expected store.remove("Talisker", 50) to be called once, actually never
  called
    param  _1 == Talisker
    param  _2 == 50
  "

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 1 | 1 failed
```

Test by setting up expectations

```
class mock_store : public store {
publi
  MAK                                                    override);
  MAK                                                    erride);
};
TEST_
{
  orc
  tes
  moc
  {
    t
    R
      REQUIRE_CALL(store, remove("Talisker", 50))
        .IN_SEQUENCE(seq);
      test_order.fill(store);
  }
}
```

```
class order
{
public:
  void add(const std::string& name, size_t s) {
    article = name;
    quantity = s;
  }
  void fill(store& the_store) {
    if (the_store.inventory(article) >= quantity) {
      the_store.remove(article, quantity);
    }
  }
private:
  std::string article;
  size_t quantity;
};
```

Test by setting up expectations

```
class mock_store : public store {
publi
  MAK                                                    override);
  MAK                                                    erride);
};
TEST_
{
  orc
  tes
  moc
  {
    t
    E
  }
}
```

```
class order
{
public:
  void add(const std::string& name, size_t s) {
    article = name;
    quantity = s;
  }
  void fill(store& the_store) {
    if (the_store.inventory(article) >= quantity) {
      the_store.remove(article, quantity);
    }
  }
private:
  std::string article;
  size_t quantity;
};
```

```
===============================================================================
test cases: 2 | 2 passed
assertions: - none -
```

- **Background**
- **Adaptation to unit test framework**
- **Make mock member functions**
- **`override`**
- **`REQUIRE_CALL`**
- **Expectation objects**
- **Lifetime of expectation**
- **`RETURN`**
- **Sequence control**
- Templated type
- Wildcard and `WITH`
- Positional parameter names

https://www.instagram.com/p/BRbWpWXhyhM/

- `ALLOW_CALL`
- `TIMES`
- Print custom data types
- Named expectations
- `SIDE_EFFECT`
- `LR_` prefix
- `FORBID_CALL`
- Callbacks
- Trompeloeil matchers
- Writing own matchers
- Lifetime control
- Advanced sequence control

Improvisation around **http://martinfowler.com/articles/mocksArentStubs.html**

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

struct mock_store : store {


};
```

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

struct mock_store : store {


};
```

This API is no good
if there may be several
orders handled
simultaneously

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

struct mock_store : store {


};
```

And what if we want another type for the article identification?

This API is no good if there may be several orders handled simultaneously

```
class store {
public:
  virtual ~store() = default;
  virtual size_t inventory(const std::string& article) const = 0;
  virtual void remove(const std::string& article, size_t quantity) = 0;
};

struct mock_store : store {


};
```

And is an OO design with a pure abstract base class really what we want?

And what if we want another type for the article identification?

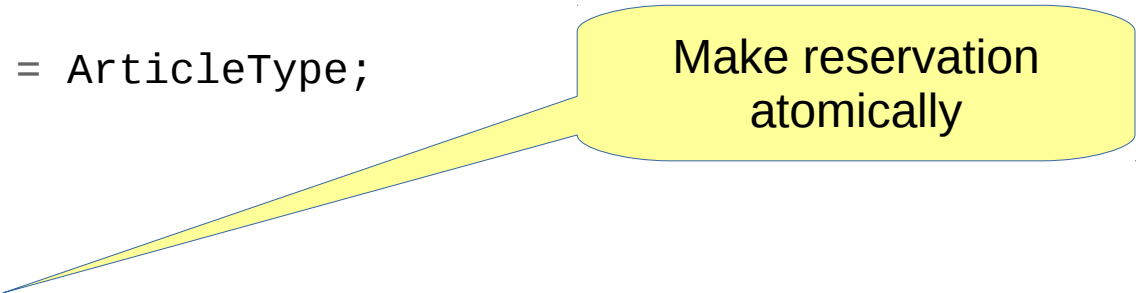
This API is no good if there may be several orders handled simultaneously

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;
```

Allow arbitrary types for article identification

```
  MAKE_MOCK2(reserve, size_t(const article_type&, size_t));
  MAKE_MOCK2(cancel, void(const article_type&, size_t));
  MAKE_MOCK2(remove, void(const article_type&, size_t));
};

using whisky_store = mock_store<std::string>;
```

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;

  MAKE_MOCK2(reserve, size_t(const article_type&, size_t));
  MAKE_MOCK2(cancel, void(const article_type&, size_t));
  MAKE_MOCK2(remove, void(const article_type&, size_t));
};

using whisky_store = mock_store<std::string>;
```



```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;



  MAKE_MOCK2(reserve, size_t(const article_type&, size_t));
  MAKE_MOCK2(cancel, void(const article_type&, size_t));
  MAKE_MOCK2(remove, void(const article_type&, size_t));
};

using whisky_store = mock_store<std::string>;
```

Cancel a reservation

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;



  MAKE_MOCK2(reserve, size_t(const article_type&, size_t));
  MAKE_MOCK2(cancel, void(const article_type&, size_t));
  MAKE_MOCK2(remove, void(const article_type&, size_t));
};

using whisky_store = mock_store<std::string>;
```

Remove from the store what you have reserved

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;




  MAKE_MOCK2(reserve, size_t(const article_type&, size_t));
  MAKE_MOCK2(cancel, void(const article_type&, size_t));
  MAKE_MOCK2(remove, void(const article_type&, size_t));
};

using whisky_store = mock_store<std::string>;
```

And a convenience when writing the tests.

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;
  struct record {
    article_type article;
    size_t quantity;
  };
  MAKE_MOCK1(reserve, size_t(const record&));
  MAKE_MOCK1(cancel, void(const record&));
  MAKE_MOCK1(remove, void(const record&));
};

using whisky_store = mock_store<std::string>;
using record = whisky_store::record;
```

Reduces repetition

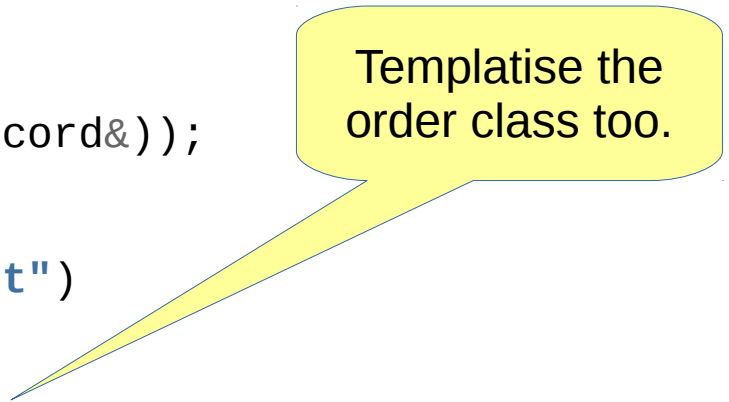
```
template <typename ArticleType>
struct mock_store {
public:
  using whisky_store = store<std::string>;
  whisky_store& a_store = ...
  order<whisky_store> the_order(a_store);
  if (the_order.add({"Talisker", 50}) == 50)
    the_order.fill();
  };
  MAK
  MAKE_MOCK1(cancel, void(const record&));
  MAKE_MOCK1(remove, void(const record&));
};

using whisky_store = mock_store<std::string>;
using record = whisky_store::record;
```

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {

    REQUIRE_CALL(store, reserve(record{"Talisker", 51}))
      .RETURN(50);

    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

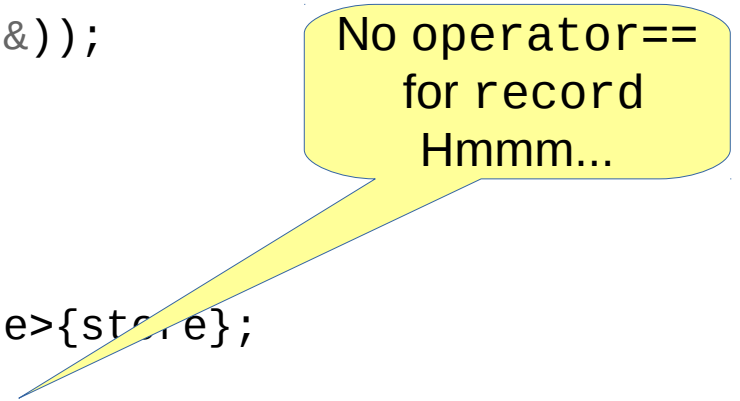
Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {

    REQUIRE_CALL(store, reserve(record{"Talisker", 51}))
      .RETURN(50);

    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```



Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {

    REQUIRE_CALL(store, reserve(record{"Talisker", 51}))
      .RETURN(50);

    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```



Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    using trompeloeil::_;                    Matcher for any value and any type
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    using trompeloeil::_;
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

So accept call with any record

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<wh
  {
    using trompeloei
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

And then constrain it to only match the intended value

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    using trompeloeil::_;
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

Boolean expression using positional names for parameters to the function

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  struct record { ... };
  MAKE_MOCK1(reserve, size_t(const record&));
  ...
};
TEST_CASE("add returns reserved amount")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    using trompeloeil::_;
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);                                    Catch! assertion
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  str
  MAK
  ...
};
TEST_
{
  whi
  aut
  {
    using trompeloeil::_;
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  size_t add(const article_type& article, size_t quantity) {
    ...
  }
private:
  StoreType& the_store;
};
```

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  str
  MAK
  ...
};
TEST_
{
  whi
  aut
  {
    using trompeloeil::_;
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    auto q = test_order->add("Talisker", 51);
    REQUIRE(q == 50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  size_t add(const article_type& article, size_t quantity) {
    return the_store.reserve({article, quantity});
  }
private:
  StoreType& the_store;
};
```

Rewriting tests to new interface

```
template <typename ArticleType>
struct mock_store {
  str
  MAK
  ...
};
TEST_
{
  whi
  aut
  {
    using trompeloeil::_;
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  size_t add(const article_type& article, size_t quantity) {
    return the_store.reserve({article, quantity});
  }
private:
  StoreType& the_store;
};
```

```
===============================================================================
test cases: 1 | 1 passed
assertions: - none -
```

Rewriting tests to new interface

```
TEST_CASE("fill removes the reserved item")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    test_order->add("Talisker", 51);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

Rewriting tests to new interface

```
TEST_CASE("fill removes the reserved item")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    test_order->add("Talisker", 51);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

The trigger to remove from store

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    F
    t
  }
  {
    F
    t
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {

  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    R
    t
  }
  {
    R
    t
  }
  // intentionally leak order, so as not to bother with cleanup
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    F
    t
  }
  {
    F
    t
  }
  //
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
===============================================================================
All tests passed (1 assertion in 2 test cases)
```

Rewriting tests to new interface

```
TEST_CASE("destructor cancels the reserved item")
{
  whisky_store store;
  auto test_order = std::make_unique<order<whisky_store>>(store);
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    test_order->add("Talisker", 51);
  }
  {
    REQUIRE_CALL(store, cancel(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order.reset();
  }
}
```

Rewriting tests to new interface

```
TEST_CASE("destructor cancels the reserved item")
{
  whisky_store store;
  auto test_order = std::make_unique<order<whisky_store>>(store);
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 51)
      .RETURN(50);
    test_order->add("Talisker", 51);
  }
  {
    REQUIRE_CALL(store, cancel(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order.reset();
  }
}
```

Destroy the order object

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    F
    t
  }
  {
    F
    t
  }
}
```

```
Template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  ~order() {
    for (auto& line : reserved)
      the_store.cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    F
    t
  }
  {
    F
    t
  }
}
```

```
Template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  ~order() {
    for (auto& line : reserved)
      the_store.cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
===============================================================================
All tests passed (1 assertion in 3 test cases)
```

Rewriting tests to new interface

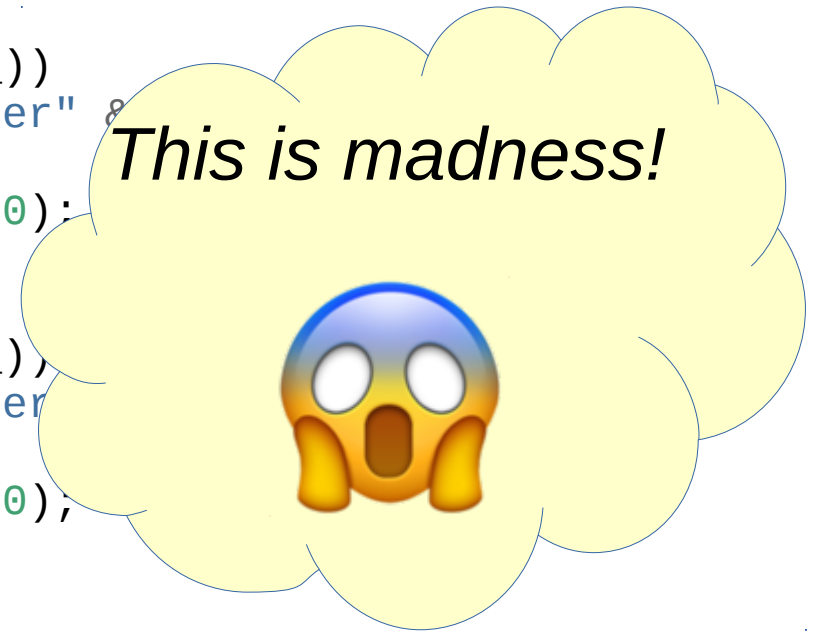
```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = std::make_unique<order<whisky_store>>(store);
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 20)
      .RETURN(20);
    test_order->add("Talisker", 20);
  }
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 31)
      .RETURN(30);
    test_order->add("Talisker", 31);
  }
  ...
}
```

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = std::make_unique<order<whisky_store>>(store);
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 20)
      .RETURN(20);
    test_order->add("Talisker", 20);
  }
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 30)
      .RETURN(30);
    test_order->add("Talisker", 30);
  }
  ...
}
```

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = std::make_unique<order<whisky_store>>(store);
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker" &
      .RETURN(20);
    test_order->add("Talisker", 20);
  }
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker
      .RETURN(30);
    test_order->add("Talisker", 30);
  }
  ...
}
```



Rewriting tests to new interface

```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    ALLOW_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker")
      .RETURN(_1.quantity);

    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    ALLOW_CALL(store, reserve(_))                    Any number of calls
      .WITH(_1.article == "Talisker")
      .RETURN(_1.quantity);

    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker")
      .TIMES(2)
      .RETURN(_1.quantity);
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Must happen exactly twice

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same art
{
  whisky_store store;
  auto test_order = new order<whisky
  {
    REQUIRE_CALL(store, res
      .WITH(_1.articl          alisker")
      .TIMES(2)
      .RETURN(_1.quantity);
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

There's also
`.TIMES(AT_LEAST(2))`

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same art
{
  whisky_store store;
  auto test_order = new order<whisky
  {
    REQUIRE_CALL(store, res
      .WITH(_1.articl                alisker")
      .TIMES(2)
      .RETURN(_1.quantity);
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

There's also
`.TIMES(AT_LEAST(2))`
and
`.TIMES(AT_MOST(5))`

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same art
{
  whisky_store store;
  auto test_order = new order<whisky
  {
    REQUIRE_CALL(store, res
      .WITH(_1.articl          alisker")
      .TIMES(2)
      .RETURN(_1.quantity);
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

There's also
`.TIMES(AT_LEAST(2))`
and
`.TIMES(AT_MOST(5))`
and even
`.TIMES(2,5)`

Rewriting tests to new interface

```
TEST_CASE("multiple adds to same article are combined")
{
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    REQUIRE_CALL(store, reserve(_))
      .WITH(_1.article == "Talisker")
      .TIMES(2)
      .RETURN(_1.quantity);
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Rew

TES

```
{
  wh
  au
  {
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a.out is a Catch v1.8.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------------
order_test2.cpp:101
.......................................................................

order_test2.cpp:133: FAILED:
explicitly with message:
  No match for call of remove with signature void(const record&) with.
    param  _1 == 40-byte object={
   0x10 0xfb 0x90 0x30 0xff 0x7f 0x00 0x00 0x80 0x00 0x00 0x00 0x00 0x00 0x00 0x00
   0x54 0x61 0x6c 0x69 0x73 0x6b 0x65 0x72 0x00 0xfb 0x90 0x30 0xff 0x7f 0x00 0x00
   0x1e 0x00 0x00 0x00 0x00 0x00 0x00 0x00 }

  Tried store.remove(_) at order_test2.cpp:113
    Failed WITH(_1.article == "Talisker" && _1.quantity == 50)

===============================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

```
  }
}
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a.out is a Catch v1.8.1 host application.
Run with -? for options

-------------------------------------------
multiple adds to same article are combined
-------------------------------------------
order_test2.cpp:101
...........................................

order_test2.cpp:133: FAILED:
explicitly with message:
  No match for call of remove with signatu  void(const record&) with.
    param  _1 == 40-byte object={
   0x10 0xfb 0x90 0x30 0xff 0x7f 0x00 0x00 0x80 0x00 0x00 0x00 0x00 0x00 0x00 0x00
   0x54 0x61 0x6c 0x69 0x73 0x6b 0x65 0x72 0x00 0xfb 0x90 0x30 0xff 0x7f 0x00 0x00
   0x1e 0x00 0x00 0x00 0x00 0x00 0x00 0x00 }

  Tried store.remove(_) at order_test2.cpp:113
    Failed WITH(_1.article == "Talisker" && _1.quantity == 50)

==========================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

Hex dump for types with no stream insertion operator. Time to implement a custom print function for **`record`**

Rewriting tests to new interface

```
namespace trompeloeil {


void print(std::ostream& os, const ::record& line)
{
  os << "{ article=" << line.article << ", quantity=" << line.quantity << " }";
}

}
```

```
TEST_
{
  whi
  aut
  {
    R
      .RETURN(_1.quantity);

    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Rew

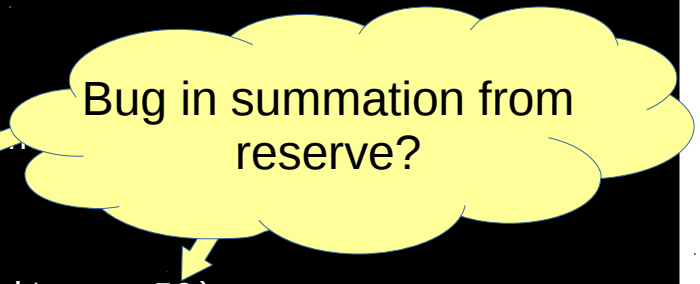
```
TEST
{
  wh
  au
  {
    test_order->fill();
  }
}
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a.out is a Catch v1.8.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------------
order_test2.cpp:109
...............................................................................

order_test2.cpp:141: FAILED:
explicitly with message:
  No match for call of remove with signature void(const record&) with.
    param  _1 == { article=Talisker, quantity=30 }

  Tried store.remove(_) at order_test2.cpp:121
    Failed WITH(_1.article == "Talisker" && _1.quantity == 50)

==========================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

Rew

```
TEST
{
  wh
  au
  {
    test_order->fill();
  }
}
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a.out is a Catch v1.8.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------------
order_test2.cpp:109
...............................................................................

order_test2.cpp:141: FAILED:
explicitly with message:
  No match for call of remove with signature void(co
    param  _1 == { article=Talisker, quantity=30 }

  Tried store.remove(_) at order_test2.cpp:121
    Failed WITH(_1.article == "Talisker" && _1.quantity == 50)

===============================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```



Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    A
    t
    t
  }
  {
    F
    t
  }
}
```

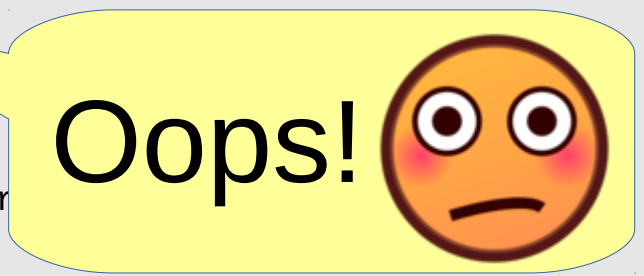
```
Template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  ~order() {
    for (auto& line : reserved)
      the_store.cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
);
```

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    A

    t
    t
  }
  {
    F

    t
  }
}
```

```
Template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  ~order() {
    for (auto& line : reserved)
      the_store.cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] = q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.fir
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```



Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    A
    t
    t
  }
  {
    F
    t
  }
}
```

```
Template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  ~order() {
    for (auto& line : reserved)
      the_store.cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] += q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
);
```

Rewriting tests to new interface

```
TEST_
{
  moc
  aut
  {
    A
    t
    t
  }
  {
    F
    t
  }
}
```

```
Template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(StoreType& s) : the_store{s} {}
  ~order() {
    for (auto& line : reserved)
      the_store.cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store.reserve({article, quantity});
    reserved[article] += q;
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store.remove({line.first, line.second});
  }
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
);
```

```
===============================================================================
All tests passed (1 assertion in 4 test cases)
```

- *Background*
- *Adaptation to unit test framework*
- *Make mock member functions*
- *`override`*
- *`REQUIRE_CALL`*
- *Expectation objects*
- *Lifetime of expectation*
- *`RETURN`*
- *Sequence control*
- **Templated type**
- **Wildcard and `WITH`**
- **Positional parameter names**

https://www.instagram.com/p/BRYcd10AJy2

- **`ALLOW_CALL`**
- **`TIMES`**
- **Print custom data types**
- Named expectations
- `SIDE_EFFECT`
- `LR_` prefix
- `FORBID_CALL`
- Callbacks
- Trompeloeil matchers
- Writing own matchers
- Lifetime control
- Advanced sequence control

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

Repetition

I want a mocked store that I can stock up at the beginning of a test, and that enforces the allowed/required behaviour of its client.

I want a mocked store that I can stock up at the beginning of a test, and that enforces the allowed/required behaviour of its client.

It is not required to handle all situations, odd cases can be handled with tests written as previously.

I want a mocked store that I can stock up at the beginning of a test, and that enforces the allowed/required behaviour of its client.

It is not required to handle all situations, odd cases can be handled with tests written as previously.

It is not required to handle several parallel orders.

I want a mocked store that I can stock up at the beginning of a test, and that enforces the allowed/required behaviour of its client.

It is not required to handle all situations, odd cases can be handled with tests written as previously.

It is not required to handle several parallel orders.

It should suffice with one map for the stock, and one map for what’s reserved by the client.

Working with data

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  {
```

```
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  {
    ALLOW_CALL(store, reserve(_))
      ...
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
}
```

Expectation must be fulfilled by the end of the scope.

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , e(NAMED_ALLOW_CALL(store, reserve(_))...)
  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> e;
}
```

std::unique_ptr<trompeloeil::expectation>

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , e(NAMED_ALLOW_CALL(store, reserve(_))
      .WITH(_1.quantity > 0 && _1.quantity <= stock[_1.article]))
  {









  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> e;
}
```

Working with data

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , e(NAMED_ALLOW_CALL(store, reserve(_))
      .WITH(_1.quantity > 0 && _1.quantity <= stock[_1.article])
      .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
      .SIDE_EFFECT(reserved[_1.article] += _1.quantity))
  {



  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> e;
}
```

Update maps

Working with data

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , e(NAMED_ALLOW_CALL(store, reserve(_))
      .WITH(_1.quantity > 0 && _1.quantity <= stock[_1.article])
      .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
      .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
      .RETURN(_1.quantity))
  {




  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> e;
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);
  {


    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {

    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {

    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

LR_ prefix means **local reference**

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {

    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

LR_ prefix means **local reference**
i.e. **s** is a reference.

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whis
  stock_w_reserve s(store, {{"Tali
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {

    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

If it wasn't already clear, this is the return expression in a lambda. **LR_** makes the capture **[&]** instead of the default **[=]**

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article ar
  whisky_store store;
  auto test_order = new order<whisky_sto
  stock_w_reserve s(store, {{"Talisker",
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {

    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {
    FORBID_CALL(store, reserve(_));
    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {
    FORBID_CALL(store, reserve(_));
    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

Adds entry first in expectation list for `reserve(const record&)`

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string,
  std::map<std::string, size_t> reserve
}
TEST_CASE("multiple adds to same articl
  whisky_store store;
  auto test_order = new order<whisky_store>(
  stock_w_reserve s(store, {{"Talisker",50}});
  test_order->add("Talisker", 20);
  test_order->add("Talisker", 30);

  {
    FORBID_CALL(store, reserve(_));
    REQUIRE_CALL(store, remove(_))
      .LR_WITH(s.reserved[_1.article] == _1.quantity);
    test_order->fill();
  }
}
```

Multiple expectations on the **same object and same function are tried in reverse order of creation**.
reserve( ) is already allowed from `stock_w_reserve`,
but this unconstrained expectation match first, so errors are caught.

Adds entry first in expectation list for `reserve(const record&)`

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    stock_w_reserve s(store, {{"Talisker",50}});
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article] == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Or maybe better to only allow reserve in local scope?

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple adds to same article are combined") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  {
    stock_w_reserve s(store, {{"Talisker",50}});
    test_order->add("Talisker", 20);
    test_order->add("Talisker", 30);
  }
  {
    REQUIRE_CALL(store, remove(_))
      .WITH(_1.article] == "Talisker" && _1.quantity == 50);
    test_order->fill();
  }
}
```

Working with data

GIMME MOAR WHISKY!!!


```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> st
  std::map<std::string, size_t> reserved;
};
TEST_CASE("multiple articles can be ordered") {
  whisky_store store;
  stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
  auto test_order = new order<whisky_store>{store};
  test_order->add("Oban", 5);
  test_order->add("Talisker", 30);
  {

    ALLOW_CALL(store, remove(_))
    .LR_WITH(s.reserved[_1.name] == _1.quantity);
    .LR_SIDE_EFFECT(s.reserved.erase(_1.name));
    test_order->fill();
  }
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple articles can be ordered") {
  whisky_store store;
  stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
  auto test_order = new order<whisky_store>{store};
  test_order->add("Oban", 5);
  test_order->add("Talisker", 30);
  {

    ALLOW_CALL(store, remove(_))
    .LR_WITH(s.reserved[_1.name] == _1.quantity);
    .LR_SIDE_EFFECT(s.reserved.erase(_1.name));
    test_order->fill();
  }
}
```

Check removal of all

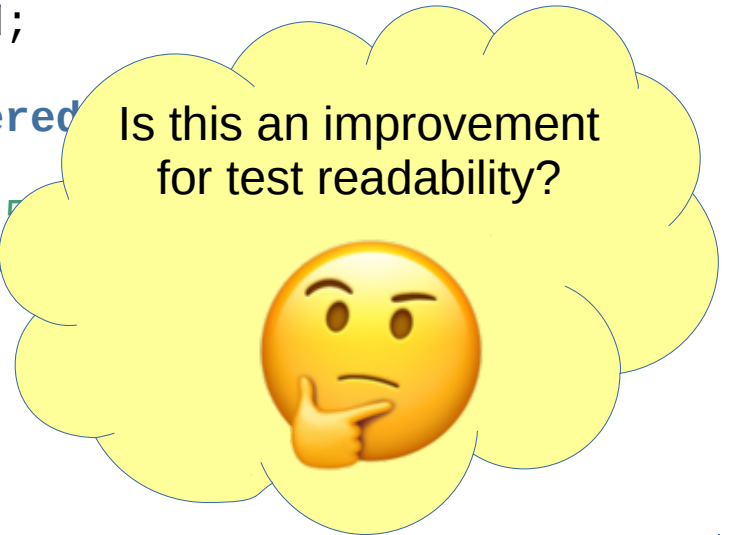
```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
}
TEST_CASE("multiple articles can be ordered") {
  whisky_store store;
  stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
  auto test_order = new order<whisky_store>{store};
  test_order->add("Oban", 5);
  test_order->add("Talisker", 30);
  {
    REQUIRE_CALL(store, remove(_))
    .TIMES(2)
    .LR_WITH(s.reserved[_1.name] == _1.quantity);
    .LR_SIDE_EFFECT(s.reserved.erase(_1.name));
    test_order->fill();
  }
}
```

Better. We expect exactly two calls

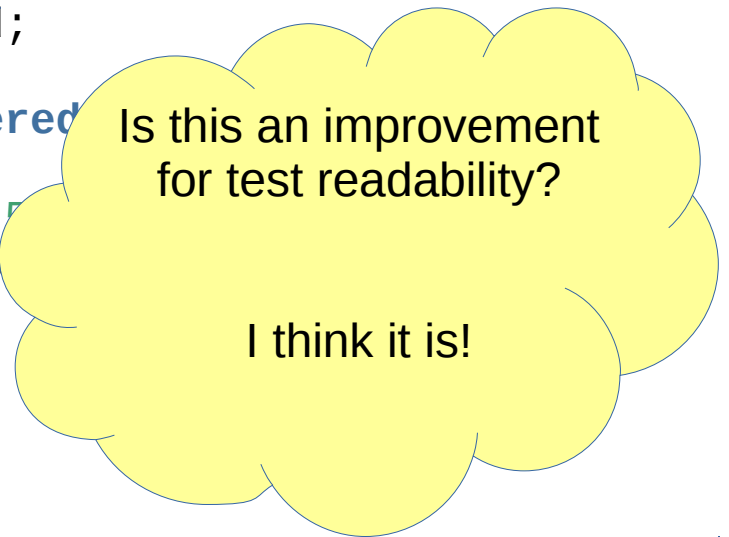
```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
};
TEST_CASE("multiple articles can be ordered") {
  whisky_store store;
  stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
  auto test_order = new order<whisky_store>{store};
  test_order->add("Oban", 5);
  test_order->add("Talisker", 30);
  {
    REQUIRE_CALL(store, remove(_
    .TIMES(2)
    .LR_WITH(s.reserved[_1.name]
    .LR_SIDE_EFFECT(s.reserved.erase(_1.name));
    test_order->fill();
  }
}
```

Should've added `REQUIRE(s.reserved.empty())` but ran out of slide space...

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
};
TEST_CASE("multiple articles can be ordered
  whisky_store store;
  stock_w_reserve s{store, {{"Talisker",
  auto test_order = new order<whisky_sto
  test_order->add("Oban", 5);
  test_order->add("Talisker", 30);
  {
    REQUIRE_CALL(store, remove(_))
    .TIMES(2)
    .LR_WITH(s.reserved[_1.name] == _1.quantity);
    .LR_SIDE_EFFECT(s.reserved.erase(_1.name));
    test_order->fill();
  }
}
```



```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store_,
                  std::map<std::string, size_t> stock_);
  std::map<std::string, size_t> reserved;
};
TEST_CASE("multiple articles can be ordered
  whisky_store store;
  stock_w_reserve s{store, {{"Talisker",
  auto test_order = new order<whisky_sto
  test_order->add("Oban", 5);
  test_order->add("Talisker", 30);
  {
    REQUIRE_CALL(store, remove(_))
    .TIMES(2)
    .LR_WITH(s.reserved[_1.name] == _1.quantity);
    .LR_SIDE_EFFECT(s.reserved.erase(_1.name));
    test_order->fill();
  }
}
```



- *Background*
- *Adaptation to unit test framework*
- *Make mock member functions*
- *`override`*
- *`REQUIRE_CALL`*
- *Expectation objects*
- *Lifetime of expectation*
- *`RETURN`*
- *Sequence control*
- *Templated type*
- *Wildcard and `WITH`*
- *Positional parameter names*

https://www.instagram.com/p/BSCuTygl7eT/

- *`ALLOW_CALL`*
- *`TIMES`*
- *Print custom data types*
- **Named expectations**
- **`SIDE_EFFECT`**
- **`LR_` prefix**
- **`FORBID_CALL`**
- Callbacks
- Trompeloeil matchers
- Writing own matchers
- Lifetime control
- Advanced sequence control

After having refactored several tests and added many new ones, a new requirement comes in.

After having refactored several tests and added many new ones, a new requirement comes in.

It must be possible to optionally get notifications through a callback when an article becomes available in stock.

After having refactored several tests and added many new ones, a new requirement comes in.

It must be possible to optionally get notifications through a callback when an article becomes available in stock.

- This should be as an optional `std::function<void()>` parameter to `add()`.

After having refactored several tests and added many new ones, a new requirement comes in.

It must be possible to optionally get notifications through a callback when an article becomes available in stock.

- This should be as an optional `std::function<void()>` parameter to `add()`.
- This implementation of `add()` must request notifications when the returned quantity is lower than the requested quantity.

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> = {})
  {
    auto q = the_store.reserve({article, quantity});
    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Afte ny new
ones

It mu ough a
callb

- Th oid()>
pa
- Th ations
wh sted
quantity.

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> = {})
  {
    auto q = the_store.reserve({article, quantity});
    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Afte ny new
ones

It mu ough a
callb

- Th oid()>
pa
- Th ations
wh sted
qu

```
===============================================================================
All tests passed (6 assertion in 6 test cases)
```

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;
  struct record {
    article_type article;
    size_t quantity;
  };

  MAKE_MOCK1(reserve, size_t(const record&));
  MAKE_MOCK1(cancel, void(const record&));
  MAKE_MOCK1(remove, void(const record&));

};

using whisky_store = mock_store<std::string>;
using record = whisky_store::record;
```

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;
  struct record {
    article_type article;
    size_t quantity;
  };
  using cb = std::function<void()>;
  MAKE_MOCK1(reserve, size_t(const record&));
  MAKE_MOCK1(cancel, void(const record&));
  MAKE_MOCK1(remove, void(const record&));
  MAKE_MOCK2(notify, void(const article_type&, cb));
};

using whisky_store = mock_store<std::string>;
using record = whisky_store::record;
```

Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  bool called = false;
  std::function<void()> callback;
  {
    stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
    REQUIRE_CALL(store, notify("Oban", _))
      .LR_SIDE_EFFECT(callback = _2);
    test_order->add("Oban", 11, [&called]() { called = true;});
  }
  callback();
  REQUIRE(called);
}
```

Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  bool called = false;
  std::function<void()> callback;
  {
    stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
    REQUIRE_CALL(store, notify("Oban", _))
      .LR_SIDE_EFFECT(callback = _2);
    test_order->add("Oban", 11, [&called]() { called = true;});
  }
  callback();
  REQUIRE(called);
}
```

Save 2nd parameter in local variable

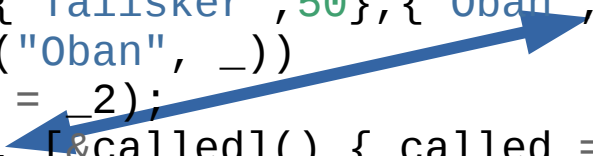
Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  bool called = false;
  std::function<void()> callback;
  {
    stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
    REQUIRE_CALL(store, notify("Oban", _))
      .LR_SIDE_EFFECT(callback = _2);
    test_order->add("Oban", 11, [&called]() { called = true;});
  }
  callback();
  REQUIRE(called);
}
```

Call with lambda that changes local variable when called.

Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  bool called = false;
  std::function<void()> callback;
  {
    stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
    REQUIRE_CALL(store, notify("Oban", _))
      .LR_SIDE_EFFECT(callback = _2);
    test_order->add("Oban", 11, [&called]() { called = true;});
  }
  callback();
  REQUIRE(called);
}
```



Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
  bool called = false;
  std::function<void()> callback;
  {
    stock_w_reserve s{store, {{"Talisker",50},{"Oban",10}}};
    REQUIRE_CALL(store, notify("Oban", _))
      .LR_SIDE_EFFECT(callback = _2);
    test_order->add("Oban", 11, [&called]() { called = true;});
  }
  callback();
  REQUIRE(called);
}
```

Ensure local variable did change after call

Adva

TES
w
s
s
a
b
s
{

}
c
R
}

```
-------------------------------------------------------------------------------
add with cb requests notification when insufficient stock
-------------------------------------------------------------------------------
order_test4.cpp:204
.....................................................................

order_test4.cpp:243: FAILED:
  CHECK( failure.empty() )
with expansion:
  false
with message:
  failure := "order_test4.cpp:214
  Unfulfilled expectation:
  Expected store.notify("Oban", _) to be called once, actually never called
    param  _1 == Oban
    param  _2 matching _
  "

order_test4.cpp:218: FAILED:
  REQUIRE( callback )
with expansion:
  false

===============================================================================
test cases: 7 | 6 passed | 1 failed
assertions: 8 | 6 passed | 2 failed
```

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> cb = {})
  {
    auto q = the_store.reserve({article, quantity});

    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> cb = {})
  {
    auto q = the_store.reserve({article, quantity});
    if (q < quantity) the_store.notify(article, cb);
    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
}
```

```
-------------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------------
order_test4.cpp:167
...............................................................................

order_test4.cpp:239: FAILED:
explicitly with message:
  No match for call of notify with signature void(const std::string&,
  std::function<void()>) with.
    param  _1 == Talisker
    param  _2 == nullptr

===============================================================================
test cases: 7 | 6 passed | 1 failed
assertions: 9 | 8 passed | 1 failed
```

Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
}
```

```
-------------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------------
order_test4.cpp:167
...............................................................................

order_test4.cpp:239: FAILED:
explicitly with message:
  No match for call of notify with signature void(const std::string&,
  std::function<void()>) with.
    param  _1 == Talisker
    param  _2 == nullptr

===============================================================================
test cases: 7 | 6 passed | 1 failed
assertions: 9 | 8 passed | 1 failed
```

Advanced usage

```
TEST_CASE("add with cb requests notification if insufficient stock") {
  whisky_store store;
  auto test_order = new order<whisky_store>{store};
```

```
-------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------
order_test4.cpp:167
.........................................

order_test4.cpp:239: FAILED:
explicitly with message:
  No match for call of notify with signature vo
  std::function<void()>) with.
    param  _1 == Talisker
    param  _2 == nullptr
}

==========================================================================
test cases: 7 | 6 passed | 1 failed
assertions: 9 | 8 passed | 1 failed
```

So the fix broke another test.

It's easy to fix, but let's think about a bigger picture for more tests.

Advanced usage

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(_))
       .WITH(_1.quantity > 0 && _1.quantity <= stock[_1.article])
       .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
       .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
       .RETURN(_1.quantity)}
     ...
  , n{NAMED_ALLOW_CALL(store, notify(_,_))
       .WITH(_2 != nullptr)}
  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store&
                  std::map<std::
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store,
       .WITH(_1.quantity > 0
       .SIDE_EFFECT(stock
       .SIDE_EFFECT(reserved[_1.
       .RETURN(_1.quantity)}
    ...
  , n{NAMED_ALLOW_CALL(store, notify(_,_))
       .WITH(_2 != nullptr)}
  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

In most tests, `notify()` is uninteresting, so we allow it as long as it follows the rules (i.e. the function is initialised with something.)

In other tests, we can set local `FORBID_CALL()` or `REQUIRE_CALL()` to enforce the rules when necessary.

Advanced usage

```
using trompeloeil::ne;
struct stock_w_reserve
{
  stock_w_reserve(whisky_
                  std::ma
  : stock(std::move(stock
  , r1{NAMED_ALLOW_CALL(s
       .WITH(_1.quantity
       .SIDE_EFFECT(stock
       .SIDE_EFFECT(reserved
       .RETURN(_1.quantity)}
    ...
  , n{NAMED_ALLOW_CALL(store, notify(_,ne(nullptr)))}

  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

ne, not-equal, here will only match calls to notify when the 2nd parameter does not compare equal to `nullptr`. The other built-in matchers are:

eq – equal to
lt – less than
le – less than or equal to
gt – greater than
ge – greater than or equal to
re – regular expression

```
using trompeloeil::ne;
struct stock_w_reserve
{
  stock_w
                                                    stock_)
  : stock
  , r1{NA
                                                    stock[_1.article])
                                                    ntity)
                                                    1.quantity)
        .RETURN(_1.quantity)}
    ...
  , n{NAMED_ALLOW_CALL(store, notify(_,ne<whisky_store::cb>(nullptr)))}

  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

By default the built-in matchers apply to any type for which the operation makes sense.

If there are conflicting overloads, an explicit type disambiguates.

Advanced usage

```
using trompeloeil::ne;
str
{
  s
  s
  s
  s
  s
  s
  s
  s
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

```
-------------------------------------------------------------------------------
multiple adds to same article are combined
-------------------------------------------------------------------------------
order_test4.cpp:169
...............................................................................

order_test4.cpp:241: FAILED:
explicitly with message:
  No match for call of notify with signature void(const std::string&,
  std::function<void()>) with.
    param  _1 == Talisker
    param  _2 == nullptr

  Tried store.notify(_,ne(nullptr)) at order_test4.cpp:73
    Expected  _2 != nullptr

===========================================================================
test cases: 7 | 6 passed | 1 failed
assertions: 9 | 8 passed | 1 failed
```

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> cb = {})
  {
    auto q = the_store.reserve({article, quantity});
    if (q < quantity) the_store.notify(article, cb);
    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
using
struc
{
  std

  : s
  , r                                                    article])

                                                         >(nullptr))}
  , r

  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> cb = {})
  {
    auto q = the_store.reserve({article, quantity});
    if (q < quantity && cb) the_store.notify(article, cb);
    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
using
struc
{
  std

  : s
  , r                                                    article])

  , r                                                    p>(nullptr))}

  {
  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, ... , n;
}
```

Advanced usage

```
template <typename StoreType>
class order
{
public:
  ...
  size_t add(
    const article_type& article,
    size_t quantity,
    std::function<void()> cb = {})
  {
    auto q = the_store.reserve({article, quantity});
    if (q < quantity && cb) the_store.notify(article, cb);
    reserved[article] += q;
    return q;
  }
  ...
private:
  StoreType& the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```
using
struc
{
  std

  : s
  , r

  , r

  {
  }
  st
  st
  std::unique_ptr<trompeloeil::expectation> r1, ... , rn;
}
```

```
article])

>(nullptr))}
```

```
===============================================================================
All tests passed (8 assertion in 7 test cases)
```

- *Background*
- *Adaptation to unit test framework*
- *Make mock member functions*
- *`override`*
- *`REQUIRE_CALL`*
- *Expectation objects*
- *Lifetime of expectation*
- *`RETURN`*
- *Sequence control*
- *Templated type*
- *Wildcard and `WITH`*
- *Positional parameter names*

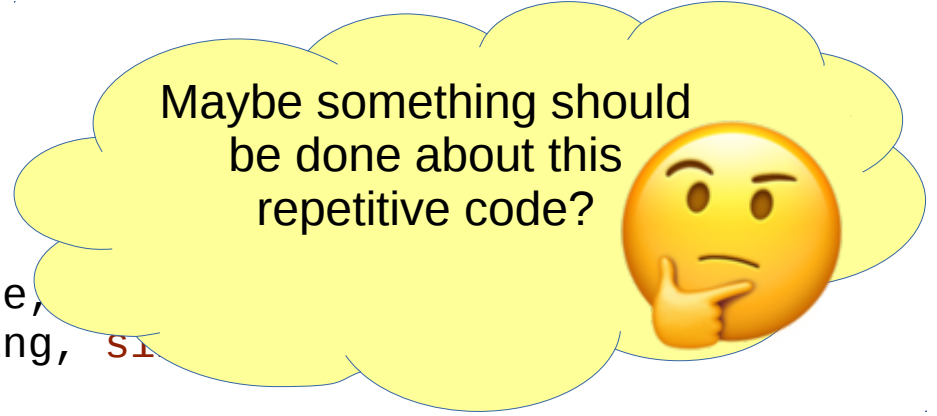
https://www.instagram.com/p/BR-q9QUhrCz

- *`ALLOW_CALL`*
- *`TIMES`*
- *Print custom data types*
- *Named expectations*
- *`SIDE_EFFECT`*
- *`LR_` prefix*
- *`FORBID_CALL`*
- **Callbacks**
- **Trompeloeil matchers**
- Writing own matchers
- Lifetime control
- Advanced sequence control

Matchers

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(_))
        .WITH(stock[_1.article] >= _1.quantity)
        .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
        .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
        .RETURN(_1.quantity)}
  , r2{NAMED_ALLOW_CALL(store, reserve(_))
         .WITH(stock[_1.article] < _1.quantity)
       ...}
  , n{NAMED_ALLOW_CALL(store, notify(_,ne(nullptr)))}
  {  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, r2 , n;
}
```

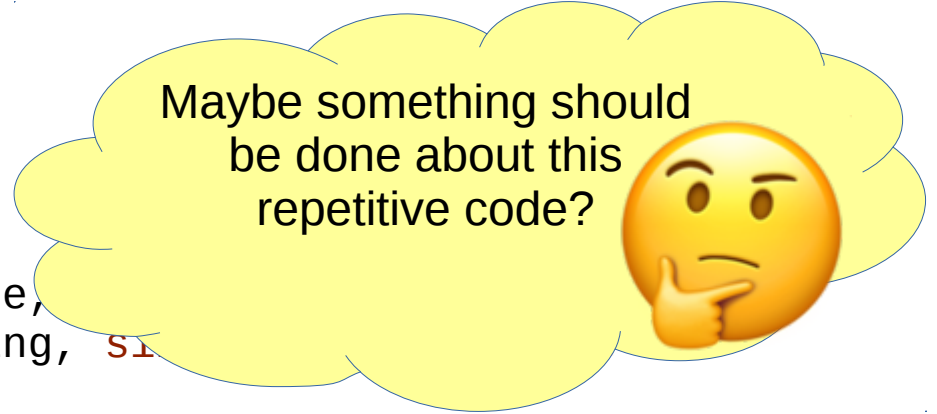
Matchers

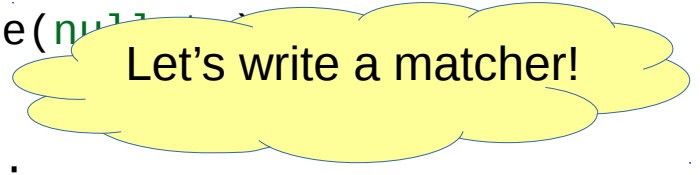



```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, si
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(_))
        .WITH(stock[_1.article] >= _1.quantity)
        .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
        .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
        .RETURN(_1.quantity)}
  , r2{NAMED_ALLOW_CALL(store, reserve(_))
         .WITH(stock[_1.article] < _1.quantity)
       ...}
  , n{NAMED_ALLOW_CALL(store, notify(_,ne(nullptr)))}
  {  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, r2 , n;
}
```

Matchers

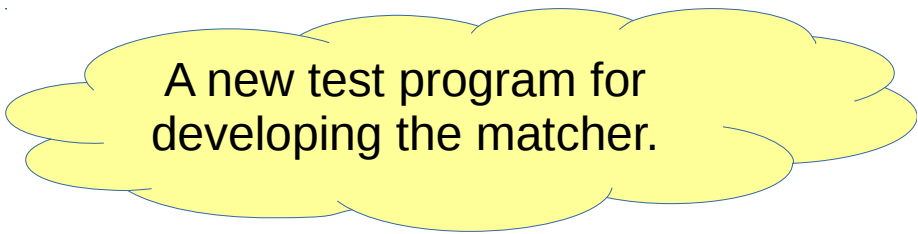
```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, si
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(_))
        .WITH(stock[_1.article] >= _1.quantity)
        .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
        .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
        .RETURN(_1.quantity)}
  , r2{NAMED_ALLOW_CALL(store, reserve(_))
         .WITH(stock[_1.article] < _1.quantity)
       ...}
  , n{NAMED_ALLOW_CALL(store, notify(_,ne(nu
  {  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, r2 , n;
}
```





Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      "something something something";

    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```



Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      "something something something";

    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```

A mock object to test run the matcher on

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      "something something something";

    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      "something something something";

    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```

Usage looks good!

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {


    FAIL("what() == " << e.what());

  }
}
```

Rely on default reporting by throwing exception

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20}};
  try {
    REQUIRE_CALL(s, func(a
    s.func({"Laphroaig", 1
    FAIL("was wrongly acce
  }
  catch (std::exception& e) {


    FAIL("what() == " << e.what());

  }
}
```

My standard technique is to deliberately fail, so I can see the message, until I'm happy with it, and then encode it in a regular expression.

Matchers

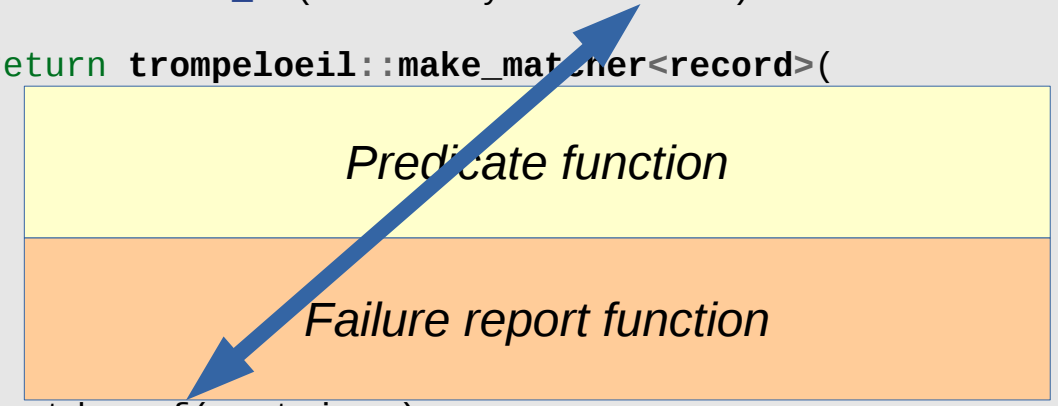
```
struc
  MAK
};
using
TEST_
  S s
  inv
  try
    F
    s
    F
  }
  cat
    F
  }
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(inventory& container)
{
  return trompeloeil::make_matcher<record>(
    Predicate function
    Failure report function
    Data shared between the two above
  );
}
```

Matchers

```
struc
  MAK
};
using
TEST_
  S s
  inv
  try
    R
    s
    R
  }
  cat
    R
  }
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(inventory& container)
{
  return trompeloeil::make_matcher<record>(
    std::ref(container)
  );
}
```



Matchers

```
struc
  MAK
};
using
TEST_
  S s
  inv
  try
    R
    s
    R
  }
  cat

    R

  }
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {

      return false;
    },
    std::ref(container)
  );
}
```

Failure report function

Matchers

```
struc
  MAK
};
using
TEST_
  S s
  inv
  try
    R
    s
    R
  }
  cat

    R

  }
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {

      return false;
    },
    [](std::ostream& os, const inventory& c) {
      os << " in ";
      trompeloeil::print(os, c);
    },
    std::ref(container)
  );
}
```

Matchers

```
struc
  MAK
};
using
TEST_
  S s
  inv
  try
    F
    s
    F
  }
  cat
    F
  }
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {

      return false;
    },
    [](std::ostream& os, const inventory& c) {
      os << " in ";
      trompeloeil::print(os, c);
    },
    std::ref(container)
  );
}
```

`trompeloeil::print()` does a comma separated member by member print of anything that has a `.begin()` and `.end()`. It also does element by element print of `std::pair<>` and `std::tuple<>`. And it does so recursively

Matchers

```
using inventory = std::map<std::string, size_t>;
```

```
struc
  MA
};
us
TE
  s
  i
  t
  }
  c
  }
}
```

```
-------------------------------------------------------------------------------
record with unknown article fails
-------------------------------------------------------------------------------
matcher_test.cpp:67
...............................................................................

matcher_test.cpp:79: FAILED:
explicitly with message:
  what() ==
  No match for call of func with signature void(const record&) with.
    param  _1 == { article=Laphroaig, quantity=1 }

  Tried s.func(available_in(stock)) at matcher_test.cpp:72
    Expected  _1 in { { Oban, 20 }, { Talisker, 50 } }

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      ".*_1 in \\{ \\{ Oban, 20 \\}, \\{ Talisker, 50 \\} \\}.*";

    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};
using inventory = std::map<std::string, size_t>;
TEST_CASE("record with article not in stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Laphroaig", 1});
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      ".*_1 in \\{ \\{ Oban, 20 \\}, \\{ Talisker, 50 \\} \\}.*";

    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```

```
===============================================================================
All tests passed (1 assertion in 1 test case)
```

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};

TEST_CASE("record with article and quantity in stock is accepted")
{
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  REQUIRE_CALL(s, func(available_in(stock)));
  s.func({"Talisker", 50});
}
```

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};

TE
{
  S
  I
  R
  s
}
```

```
-------------------------------------------------------------------------------
record with article and quantity in stock is accepted
-------------------------------------------------------------------------------
matcher_test.cpp:42
...............................................................................

matcher_test.cpp:42: FAILED:
due to unexpected exception with message:

  No match for call of func with signature void(const record&) with.
    param  _1 == { article=Talisker, quantity=50 }

  Tried s.func(available_in(stock)) at matcher_test.cpp:46
    Expected  _1 in { { Oban, 20 }, { Talisker, 50 } }

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

Matchers

```
struc
  MAK
};

TEST_
{
  S s
  inv
  REQ
  s.f
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(const inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {
      return c.find(value.article) != c.end();

    },
    [](std::ostream& os, const inv
      os << " in “;
      trompeloeil::print(os, c)
    },
    std::ref(container)
  );
}
```

```
ccepted")
```

Make the predicate an actual check of a condition.

Matchers

```
struc
  MAK
};

TEST_
{
  S s
  inv
  REC
  s.f
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(const inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {
      return c.find(value.article) != c.end();

    },
    [](std::ostream& os, const inventory& c) {
      os << " in “;
      trompeloeil::print(os, c);
    },
    std::ref(container)
  );
}
```

```
ccepted")
```

```
===============================================================================
All tests passed (1 assertion in 2 test case)
```

Matchers

```
struct S {
  MAKE_MOCK1(func, void(const record&));
};

TEST_CASE("record with quantity exceeding stock fails") {
  S s;
  inventory stock{{"Talisker", 50}, {"Oban", 20 } };
  try {
    REQUIRE_CALL(s, func(available_in(stock)));
    s.func({"Oban", 21});                                  One too many
    FAIL("was wrongly accepted");
  }
  catch (std::exception& e) {
    auto re =
      ".*_1 in \\{ \\{ Oban, 20 \\}, \\{ Talisker, 50 \\} \\}.*";
    INFO("what() == " << e.what());
    REQUIRE(std::regex_search(e.what(), std::regex(re)));
  }
}
```

Matchers

```
using inventory = std::map<std::string, size_t>;

auto available_in(const inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {
      auto i = c.find(value.article);
      return i != c.end() && value.quantity <= i->second;
    },
    [](std::ostream& os, const inventory& c) {
      os << " in ";
      trompeloeil::print(os, c);
    },
    std::ref(container)
  );
}
```

```
struc
  MAK
};

TEST_
  S s
  inv
  try
    R
    s
    R
  }
  cat
    a
                                                                    \\}.*";
    I
    R
  }
}
```

Matchers

```
struc
  MAK
};

TEST_
  S s
  inv
  try
    R
    s
    R
  }
  cat
    a
                                                                    \\}.*";
    I
    R
  }
}
```

```
using inventory = std::map<std::string, size_t>;

auto available_in(const inventory& container)
{
  return trompeloeil::make_matcher<record>(
    [](const record& value, const inventory& c) {
      auto i = c.find(value.article);
      return i != c.end() && value.quantity <= i->second;
    },
    [](std::ostream& os, const inventory& c) {
      os << " in ";
      trompeloeil::print(os, c);
    },
    std::ref(container)
  );
}
```

```
===============================================================================
All tests passed (1 assertion in 3 test case)
```

Matchers

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(available_in(stock)))
       .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
       .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
       .RETURN(_1.quantity)}
  , r2{NAMED_ALLOW_CALL(store, reserve(_))
        .WITH(stock[_1.article] < _1.quantity)
      ...}
  , n{NAMED_ALLOW_CALL(store, notify(_,ne(nullptr)))}
  {  }
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, r2 , n;
}
```

Matchers

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(available_in(stock)))
       .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
       .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
       .RETURN(_1.quantity)}
  , r2{NAMED_ALLOW_CALL(store, reserve(!available_in(stock)))
       ...}
  ...
  , n{NAMED_ALLOW_CALL(store, notify(_,ne(nullptr)))}
  {...}
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, r2 , n;
}
```

Matchers can be negated using the logical-not operator (!)

Matchers

```
struct stock_w_reserve
{
  stock_w_reserve(whisky_store& store,
                  std::map<std::string, size_t> stock_)
  : stock(std::move(stock_))
  , r1{NAMED_ALLOW_CALL(store, reserve(available_in(stock)))
       .SIDE_EFFECT(stock[_1.article] -= _1.quantity)
       .SIDE_EFFECT(reserved[_1.article] += _1.quantity)
       .RETURN(_1.quantity)}
  , r2{NAMED_ALLOW_CALL(store, reserve(!available_in(stock)))
       ...}
                                          otify(_,ne(nullptr)))}
  std::map<std::string, size_t> stock;
  std::map<std::string, size_t> reserved;
  std::unique_ptr<trompeloeil::expectation> r1, r2 , n;
}
```

Matchers can be negated using the logical-not operator (!)

- *Background*
- *Adaptation to unit test framework*
- *Make mock member functions*
- *`override`*
- *`REQUIRE_CALL`*
- *Expectation objects*
- *Lifetime of expectation*
- *`RETURN`*
- *Sequence control*
- *Templated type*
- *Wildcard and `WITH`*
- *Positional parameter names*

https://www.instagram.com/p/BR-QInAAqQi

- *`ALLOW_CALL`*
- *`TIMES`*
- *Print custom data types*
- *Named expectations*
- *`SIDE_EFFECT`*
- *`LR_` prefix*
- *`FORBID_CALL`*
- *Callbacks*
- *Trompeloeil matchers*
- **Writing own matchers**
- Lifetime control
- Advanced sequence control

After having refactored several tests, a new requirement comes in again.

The `order` class must accept ownership of the `store` instance, and after **`fill()`** it must destroy it.

```cpp
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(std::unique_ptr<StoreType> s) : the_store{std::move(s)} {}
  ~order() {
    for (auto& line : reserved)
      the_store->cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store->reserve({article, quantity});
    reserved[article] += q;
    if (q < quantity && cb) the_store->notify(name, cb);
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store->remove({line.first, line.second});
  }
private:
  std::unique_ptr<StoreType> the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

```cpp
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreTyp
  order(std::unique_ptr<StoreType> s) :
  ~order() {
    for (auto& line : reserved)
      the_store->cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store->reserve({article, quantity});
    reserved[article] += q;
    if (q < quantity && cb) the_store->notify(name, cb);
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store->remove({line.first, line.second});
  }
private:
  std::unique_ptr<StoreType> the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

So far a trivial change with very minor impact on the test code.

But how to test the destruction on `fill()`?

Lifetime management

```
TEST_CASE("store is destroyed after fill")
{
  auto store = new trompeloeil::deathwatched<whisky_store>;
  order<whisky_store> test_order{std::unique_ptr<whisky_store>(store)};
  {
    stock_w_reserve s{*store, {{"Talisker", 10}, {"Oban", 20}}};
    test_order.add("Oban", 5);
    test_order.add("Talisker", 5);
  }
  {


    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5);

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1.quantity == 5);
    ...
  }
}
```

A `deathwatched` object is not allowed to be destroyed until we tell it to.

And when we tell it to, it **must** die.

Lifetime management

```
TEST_CASE("store is destroyed after fill")
{
  auto store = new trompeloeil::deathwatched<whisky_store>;
  ...
    test_order.add("Talisker", 5);
  }
  {


    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5);

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1.quantity == 5);

    REQUIRE_DESTRUCTION(*store);
    test_order.fill();
  }
}
```

No, Mr. Bond,
I expect you to die!

Lifetime management

```
TEST_CASE("store is destroyed after fill")
{
  auto store = new trompeloeil::deathwatched<whisky_store>;
  ...
    test_order.add("Talisker", 5);
  }
```

```
In file included from order_test7.cpp:1:0:
/home/bjorn/devel/trompeloeil/trompeloeil.hpp: In instantiation of 'class
trompeloeil::deathwatched<mock_store<std::basic_string<char> > >':
order_test7.cpp:259:33:   required from here
/home/bjorn/devel/trompeloeil/trompeloeil.hpp:1878:5: error: static assertion failed:
virtual destructor is a necessity for deathwatched to work
     static_assert(std::has_virtual_destructor<T>::value,
     ^
```

```
      .WITH(_1.article == "Oban" && _1.quantity == 5);

    REQUIRE_DESTRUCTION(*store);

    test_order.fill();
  }
}
```

No, Mr. Bond,
I expect you to die!

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;
  struct record {
    article_type article;
    size_t quantity;
  };
  using callback = std::function<void()>;

  MAKE_MOCK1(reserve, size_t(const record&));
  MAKE_MOCK1(cancel, void(const record&));
  MAKE_MOCK1(remove, void(const record&));
  MAKE_MOCK2(notify, void(const article_type&, callback));
};

using whisky_store = mock_store<std::string>;
using record = whisky_store::record;
```

```
template <typename ArticleType>
struct mock_store {
public:
  using article_type = ArticleType;
  struct record {
    article_type article;
    size_t quantity;
  };
  using callback = std::function<void()>;
  virtual ~mock_store() = default;
  MAKE_MOCK1(reserve, size_t(const record&));
  MAKE_MOCK1(cancel, void(const record&));
  MAKE_MOCK1(remove, void(const record&));
  MAKE_MOCK2(notify, void(const article_type&, callback));
};

using whisky_store = mock_store<std::string>;
using record = whisky_store::record;
```

Add virtual destructor

```
-------------------------------------------------------------------------------
store is deleted after fill
-------------------------------------------------------------------------------
order_test6.cpp:263
...............................................................................

order_test6.cpp:308: FAILED:
  CHECK( failure.empty() )
with expansion:
  false
with message:
  failure := "order_test6.cpp:283
  Object *store is still alive"

order_test6.cpp:303: FAILED:
explicitly with message:
  No match for call of cancel with signature void(const Record&) with.
    param  _1 == { article=Talisker, quantity=5 }

terminate called after throwing an instance of 'Catch::TestFailureException'
order_test6.cpp:263: FAILED:
  {Unknown expression after the reported line}
with expansion:

due to a fatal error condition:
  SIGABRT - Abort (abnormal termination) signal

===============================================================================
test cases:  9 | 8 passed | 1 failed
assertions: 11 | 8 passed | 3 failed
```

Failure as expected.

Lifetim

```
TEST_C
{
  ...
    te
    te
  }
  {


    RE


    RE


    RE


    te
  }
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(std::unique_ptr<StoreType> s) : the_store{std::move(s)} {}
  ~order() {
    for (auto& line : reserved)
      the_store->cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store->reserve({article, quantity});
    reserved[article] += q;
    if (q < quantity && cb) the_store->notify(name, cb);
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store->remove({line.first, line.second});
    reserved.clear();

  }
private:
  std::unique_ptr<StoreType> the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Mr. Bond,
ou to die!

Lifetim

```
TEST_C
{
  ...
    te
    te
  }
  {


    RE


    RE


    RE


    te
  }
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(std::unique_ptr<StoreType> s) : the_store{std::move(s)} {}
  ~order() {
    for (auto& line : reserved)
      the_store->cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store->reserve({article, quantity});
    reserved[article] += q;
    if (q < quantity && cb) the_store->notify(name, cb);
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store->remove({line.first, line.second});
    reserved.clear();
    the_store.reset();
  }
private:
  std::unique_ptr<StoreType> the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Mr. Bond,
ou to die!

Lifetim

```
TEST_C
{
   ...
      te
      te
   }
   {


      RE


      RE


      RE


      te
   }
}
```

```
template <typename StoreType>
class order
{
public:
  using article_type = typename StoreType::article_type;
  order(std::unique_ptr<StoreType> s) : the_store{std::move(s)} {}
  ~order() {
    for (auto& line : reserved)
      the_store->cancel({line.first, line.second});
  }
  size_t add(const article_type& article, size_t quantity) {
    auto q = the_store->reserve({article, quantity});
    reserved[article] += q;
    if (q < quantity && cb) the_store->notify(name, cb);
    return q;
  }
  void fill() {
    for (auto& line : reserved)
      the_store->remove({line.first, line.second});
    reserved.clear();
    the_store.reset();
  }
private:
  std::unique_ptr<StoreType> the_store;
  std::unordered_map<article_type, size_t> reserved;
};
```

Mr. Bond,
ou to die!

```
===============================================================================
All tests passed (8 assertion in 9 test case)
```

Lifetime management

```
TEST_CASE("store is destroyed after fill")
{
  ...
    test_order.add("Oban", 5);
    test_order.add("Talisker", 5);
  }
  {
    trompeloeil::sequence seq_talisker, seq_oban;

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5)
      .IN_SEQUENCE(seq_talisker);
    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1.quantity == 5)
      .IN_SEQUENCE(seq_oban);
    REQUIRE_DESTRUCTION(*store)
      .IN_SEQUENCE(seq_talisker, seq_oban);
    test_order.fill();
  }
}
```

Sequence objects are used to impose an order between otherwise unrelated expectations.

Lifetime management

```
TEST_CASE("store is destroyed after fill")
{
  ...
    test_order.add("Oban", 5);
    test_order.add("Talisker", 5);
  }
  {
    trompeloeil::sequence seq_talisker, seq_oban;

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5)
      .IN_SEQUENCE(seq_talisker);
    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1.quantity == 5)
      .IN_SEQUENCE(seq_oban);
    REQUIRE_DESTRUCTION(*store)
      .IN_SEQUENCE(seq_talisker, seq_oban);
    test_order.fill();
  }
}
```

Different sequence objects means `remove()` order is indifferent.

Lifetime management

```
TEST_CASE("store is destroyed after fill")
{
  ...
    test_order.add("Oban", 5);
    test_order.add("Talisker", 5);
  }
  {
    trompeloeil::sequence seq_talisker, seq_oban;

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5)
      .IN_SEQUENCE(seq_talisker);
    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1.quantity == 5)
      .IN_SEQUENCE(seq_oban);
    REQUIRE_DESTRUCTION(*store)
      .IN_SEQUENCE(seq_talisker, seq_oban);
    test_order.fill();
  }
}
```

Destruction uses both sequence objects, so it must be last.

Lifetime management

```
TEST_CASE("store is destroyed after fill
{
  ...
    test_order.add("Oban", 5);
    test_order.add("Talisker", 5);
  }
  {
    trompeloeil::sequence seq_talisker, seq_oban;

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5)
      .IN_SEQUENCE(seq_talisker);
    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1.quantity == 5)
      .IN_SEQUENCE(seq_oban);
    REQUIRE_DESTRUCTION(*store)
      .IN_SEQUENCE(seq_talisker, seq_oban);
    test_order.fill();
  }
}
```

The sequence objects are not needed here, though, because destruction of a mock objects with pending expectations is reported as a violation.

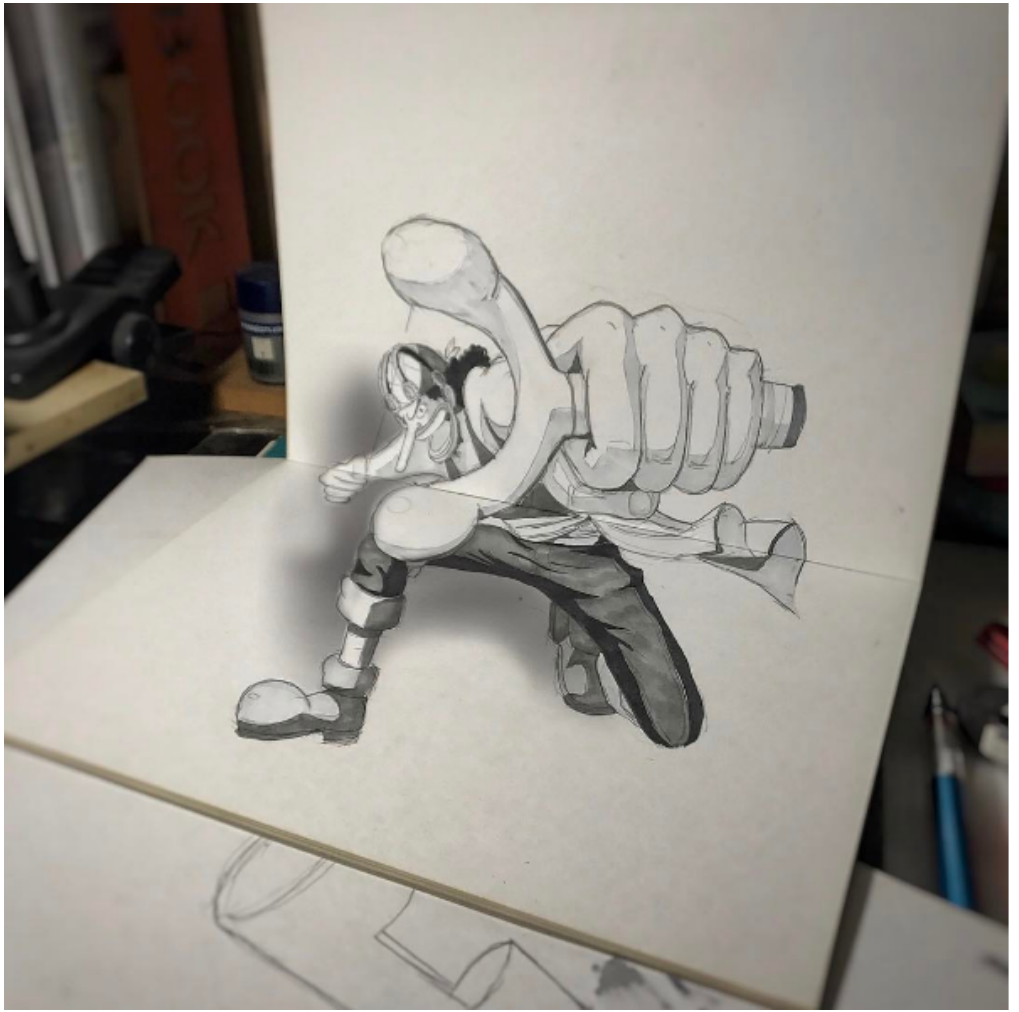
Lifetime management

```
TEST_CASE("store is destroyed after fill
{
  ...
    test_order.add("Oban", 5);
    test_order.add("Talisker", 5);
  }
  {
    trompeloeil::sequence seq_talisker, seq_oban;

    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Talisker" && _1.quantity == 5)
      .IN_SEQUENCE(seq_talisker);
    REQUIRE_CALL(*store, remove(_))
      .WITH(_1.article == "Oban" && _1
      .IN_SEQUENCE(seq_oban);
    REQUIRE_DESTRUCTION(*store)
      .IN_SEQUENCE(seq_talisker, seq_ob
    test_order.fill();
  }
}
```

The sequence objects are not needed here, though, because destruction of a mock objects with pending expectations is reported as a violation.

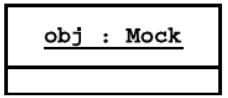
But there are other situations when sequence control is absolutely necessary. Just don't restrict too much!

- *Background*
- *Adaptation to unit test framework*
- *Make mock member functions*
- *`override`*
- *`REQUIRE_CALL`*
- *Expectation objects*
- *Lifetime of expectation*
- *`RETURN`*
- *Sequence control*
- *Templated type*
- *Wildcard and `WITH`*
- *Positional parameter names*



https://www.instagram.com/p/BSV4oxqj3cb

- *`ALLOW_CALL`*
- *`TIMES`*
- *Print custom data types*
- *Named expectations*
- *`SIDE_EFFECT`*
- *`LR_` prefix*
- *`FORBID_CALL`*
- *Callbacks*
- *Trompeloeil matchers*
- *Writing own matchers*
- **Lifetime control**
- **Advanced sequence control**



Ceci n'est pas un objet

Trompeloeil cheat sheet for implementing mock functions and placing expectations on them.

Mock implement member functions.

non-const member function	*const member function*
`MAKE_MOCKn(name, sig{, spec})`	`MAKE_CONST_MOCKn(name, sig{, spec})`

Place expectations. *Matching expectations are searched from youngest to oldest. Everything is illegal by default.*

Anonymous local object	*`std::unique_ptr<expectation>`*
`REQUIRE_CALL(obj, func(params))`	`NAMED_REQUIRE_CALL(obj, func(params))`
`ALLOW_CALL(obj, func(params))`	`NAMED_ALLOW_CALL(obj, func(params))`
`FORBID_CALL(obj, func(params))`	`NAMED_FORBID_CALL(obj, func(params))`

Refine expectations.

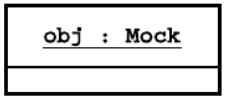
When to match

`.IN_SEQUENCE(s...)` ← Impose an ordering relation between expectations by using **sequence** objects

`.TIMES(min, max = min)` ← Define how many times an expectation must match. Default is 1. Conveniency arguments are **`AT_MOST(x)`** and **`AT_LEAST(x)`**

Parameters are _1 .. _15

Local objects are const copies		*Local objects are non-const references*
`.WITH(condition)`	← when to match →	`.LR_WITH(condition)`
`.SIDE_EFFECT(statement)`	← What to do when matching →	`.LR_SIDE_EFFECT(statement)`
`.RETURN(expression)`	← What to do when matching →	`.LR_RETURN(expression)`
`.THROW(expression)`	← What to do when matching →	`.LR_THROW(expression)`



Trompeloeil cheat sheet for matchers and object life time management.

Ceci n'est pas un objet

Matchers. Substitute for values in parameter list of expectations.

Any type allowing op				*Disambiguated type* `ANY(type)`
`_`	any value			
`eq(mark)`	value	==	mark	`eq<type>(mark)`
`ne(mark)`	value	!=	mark	`ne<type>(mark)`
`lt(mark)`	value	<	mark	`lt<type>(mark)`
`le(mark)`	value	<=	mark	`le<type>(mark)`
`gt(mark)`	value	>	mark	`gt<type>(mark)`
`ge(mark)`	value	>=	mark	`ge<type>(mark)`
`re(mark, ...)`	match regular expression /mark/			`re<type>(mark, ...)`

Use **`operator*`** to dereference pointers. E.g. **`*ne(mark)`** means parameter is pointer (like) and **`*parameter != mark`**
Use **`operator!`** to negate matchers. E.g. **`!re(mark)`** means not matching regular expression **`/mark/`**

Object life time management

`auto obj = new deathwatched<my_mock_type>(params);`
`*obj` destruction only allowed when explicitly required. Inherits from **`my_mock_type`**

Anonymous local object	*`std::unique_ptr<expectation>`*
`REQUIRE_DESTRUCTION(*obj)`	**`NAMED_REQUIRE_DESTRUCTION(*obj)`**

When to match
`.IN_SEQUENCE(s...)` ← Impose an ordering relation between expectations by using **sequence** objects

Check out some alternative mocking frameworks	
Google mock (gmock)	https://github.com/google/googletest
Mockator	http://mockator.com
FakeIt	https://github.com/eranpeer/FakeIt
HippoMocks	https://github.com/dascandy/hippomocks

Using Trompeloeil
a mocking framework for modern C++

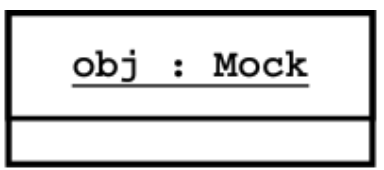
`https://github.com/rollbear/trompeloeil`

Björn Fahller

 `bjorn@fahller.se`

 `@bjorn_fahller`

 `@rollbear` *cpplang, swedencpp*



Ceci n'est pas un objet.