Mocking Modern C++ with Trompeloeil

Björn Fahller

Mocking Modern C++ with Trompeloeil

Trompe-l'œil noun (Concise Encyclopedia)

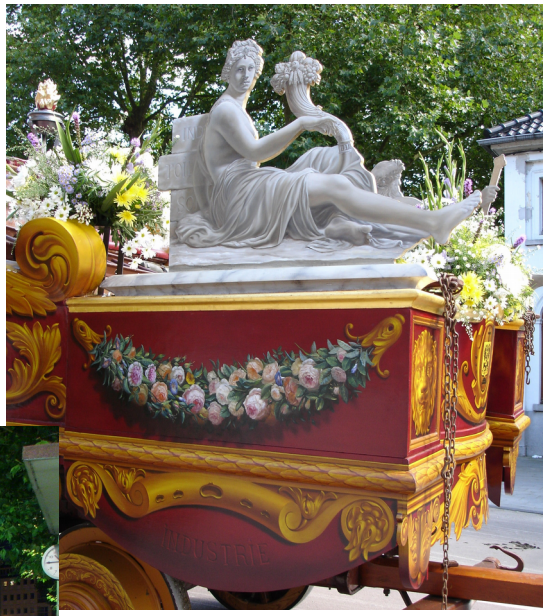
Style of representation in which a painted object is intended to deceive the viewer into believing it is the object itself...

Björn Fahller

Mocking Modern C++ with Trompeloeil

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...

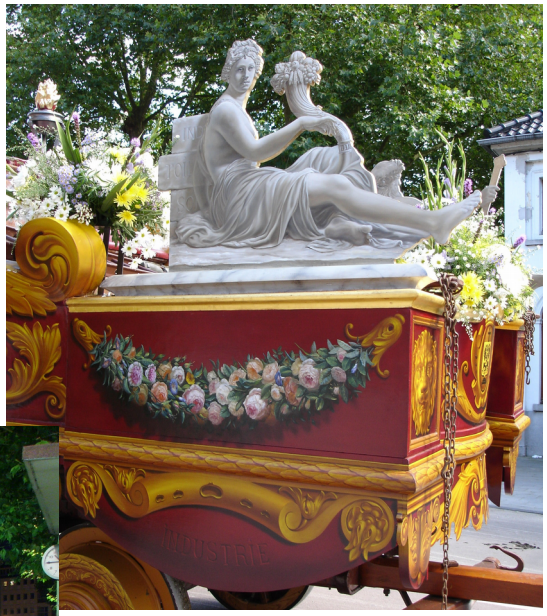


Björn Fahller

Mocking Modern C++ with Trompeloeil

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

Ceci n'est pas un objet

Mocking Modern C++ with Trompeloeil

In C++ there are two schools of mocking implementation techniques.

- Runtime manipulation of object layouts. Compiler/OS dependant implementation.

- Strict compilation within the language creating new types

Mocking Modern C++ with Trompeloeil

In C++ there are two schools of mocking implementation techniques.

- Runtime manipulation of object layouts. Compiler/OS dependant implementation.
  - Hippo mocks
  - Mockitopp
  - AMOP
  - ...

- Strict compilation within the language creating new types
  - Google mock
  - Mockator
  - Trompeloeil
  - ...

Mocking Modern C++ with Trompeloeil

Trompeloeil is:

- Pure C++14 without any dependencies
- Implemented in a single header file
- Under Boost Software License 1.0
- Adaptable to any (that I know of) unit testing framework
- Available from 

Mocking Modern C++ with Trompeloeil

Documentation

- Introduction
- Cheat Sheet (2*A4)
- Cook Book
- FAQ
- Reference

Mocking Modern C++ with Trompeloeil

Trompeloeil cook book

- Integrating with unit test frame works
- Creating Mock Classes
- Mocking private or protected member functions
- Mocking overloaded member functions
- ...

Mocking Modern C++ with Trompeloeil

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
- ...

Mocking Modern C++ with Trompeloeil

Introduction by example.

Extrapolated from Martin Fowler's whisky store order example, from the blog post "Mocks Aren't Stubs"

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

Mocking Modern C++ with Trompeloeil

Introduction by example.

Extrapolated from Martin Fowler's whisky store order example, from the blog post "Mocks Aren't Stubs"

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

```
class store {
public:
  virtual size_t inventory(const std::string& name) const = 0;
  virtual void remove(const std::string& name, size_t count) = 0;
};

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

Mocking Modern C++ with Trompeloeil

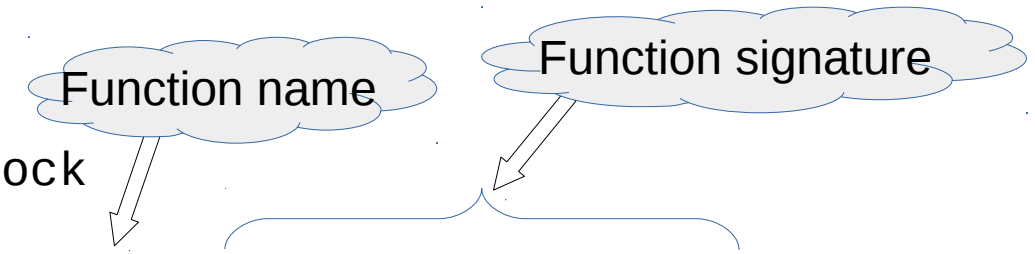
Creating a mock type.

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

Mocking Modern C++ with Trompeloeil

Creating a mock type.

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

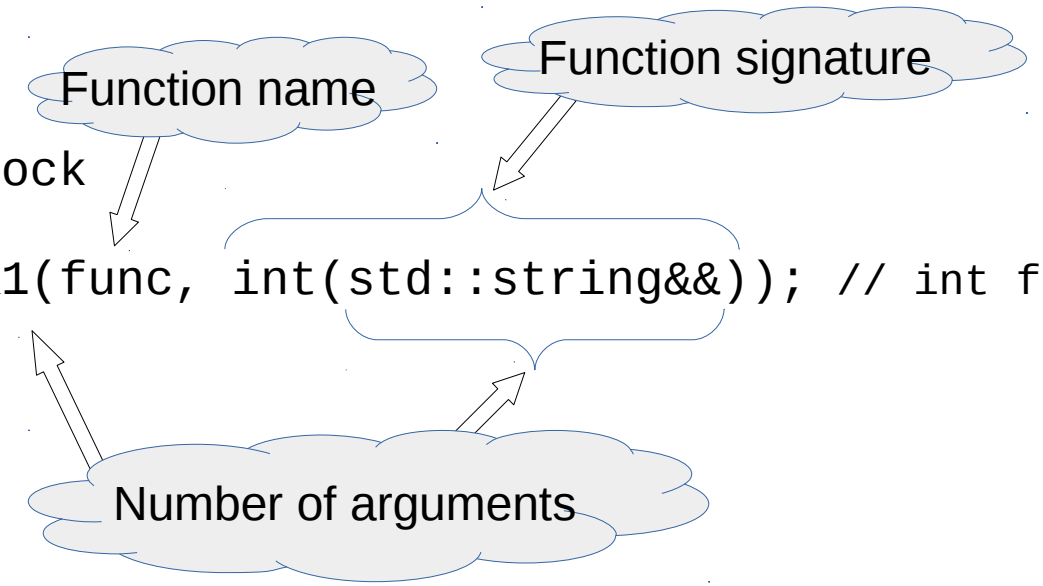



```
struct my_mock
{
  MAKE_MOCK1(func, int(std::string&&)); // int func(std::string&&);
};
```

Mocking Modern C++ with Trompeloeil

Creating a mock type.

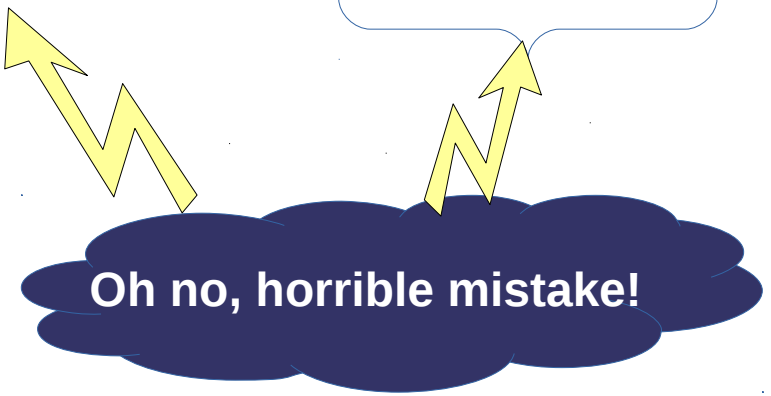
```
#include <trompeloeil.hpp>
```



Mocking Modern C++ with Trompeloeil

Creating a mock type.

```
#include <trompeloeil.hpp>

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



Mocking Modern C++ with Trompeloeil

Creating a mock type.

```
#include <trompeloeil.hpp>

struct my_mock
{
  MAKE_MOCK2(func, int(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&&));
  ^
```

Mocking Modern C++ with Trompeloeil

Creating a mock type.

```
#include <trompeloeil.hpp>

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

Full error message from g++ 5.4

```
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&&));
  ^
```

Mocking Modern C++ with Trompeloeil

Creating a mock type.

```
#include <trompeloeil.hpp>




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

Mocking Modern C++ with Trompeloeil

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&&));
};
```

Mocking Modern C++ with Trompeloeil

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&&));
};
```

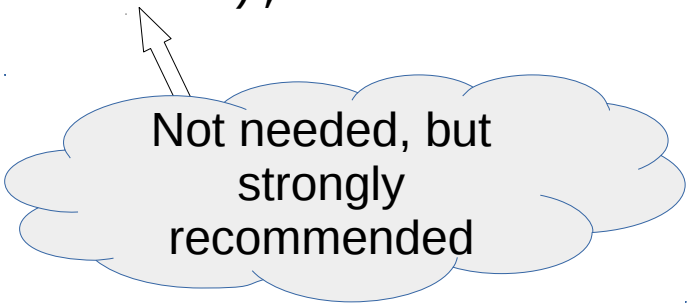
ReSharper C++
@resharper_cpp

Has a built in generator for mocks from interfaces!

Mocking Modern C++ with Trompeloeil

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);
};
```



Mocking Modern C++ with Trompeloeil

Introduction by example.

Extrapolated from Martin Fowler's whisky store order example, from the blog post "Mocks Aren't Stubs"

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

```
class store {
public:
  virtual size_t inventory(const std::string& name) const = 0;
  virtual void remove(const std::string& name, size_t count) = 0;
};

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

Mocking Modern C++ with Trompeloeil

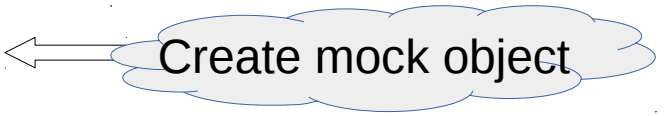
```
#include <trompeloeil.hpp>

class store {
public:
  virtual size_t inventory(const std::string& name) const = 0;
  virtual void remove(const std::string& name, size_t count) = 0;
};

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

Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

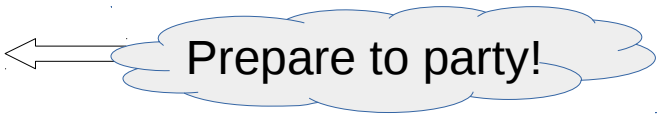
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
}
```



Mocking Modern C++ with Trompeloeil

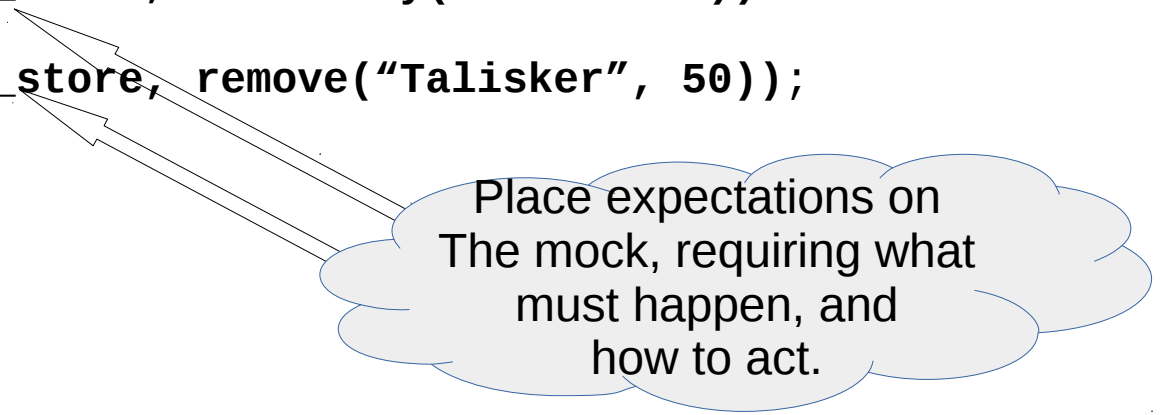
```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
```




```
}
```

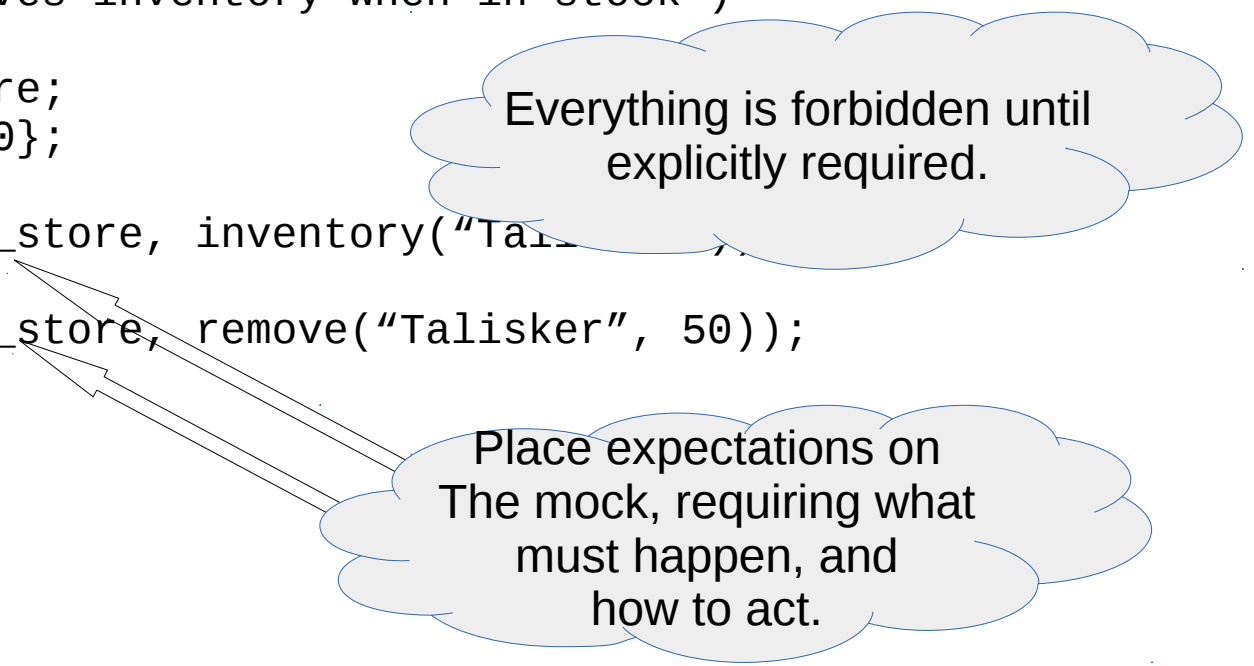
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Talisker"))
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker", 50));

  }

}
```



Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Tal
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker", 50));

  }

}
```



Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Tal
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker", 50));

  }
}
```

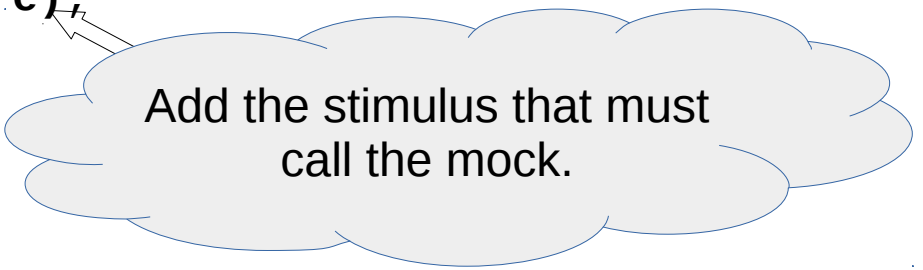
Everything is forbidden until explicitly required.

Place expectations on The mock, requiring what must happen, and how to act.

Expectations must be fulfilled at end of scope, otherwise a violation is reported.

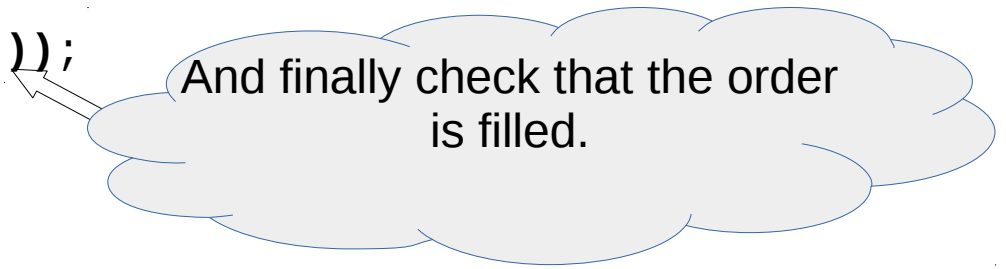
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Talisker"))
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker", 50));
    o.fill(whisky_store);
  }


}
```



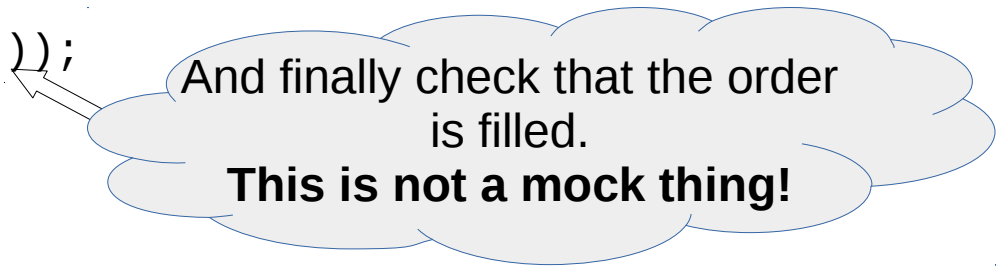
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Talisker"))
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker", 50));
    o.fill(whisky_store);
  }
  REQUIRE(o.is_filled());
}
```



Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Talisker"))
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker", 50));
    o.fill(whisky_store);
  }
  REQUIRE(o.is_filled());
}
```



Mocking Modern C++ with Trompeloeil

Result from run with empty `order` implementation.

```
failure := store_test.cpp:59
Unfulfilled expectation:
Expected whisky_store.remove("Talisker", 50) to be called once, actually
never called
  param  _1 == Talisker
  param  _2 == 50

failure := store_test.cpp:57
Unfulfilled expectation:
Expected whisky_store.inventory("Talisker") to be called once, actually never
called
  param  _1 == Talisker

store_test.cpp:62: FAILED:
  REQUIRE( o.is_filled() )
with expansion:
  false
```

Mocking Modern C++ with Trompeloeil

Let's make a sloppy `order` implementation and try again:

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.remove(what, amount); s.inventory(what);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

Mocking Modern C++ with Trompeloeil

Let’s make a sloppy `order` implementation and try again:

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.remove(what, amount); s.inventory(what);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

This passes.

Mocking Modern C++ with Trompeloeil

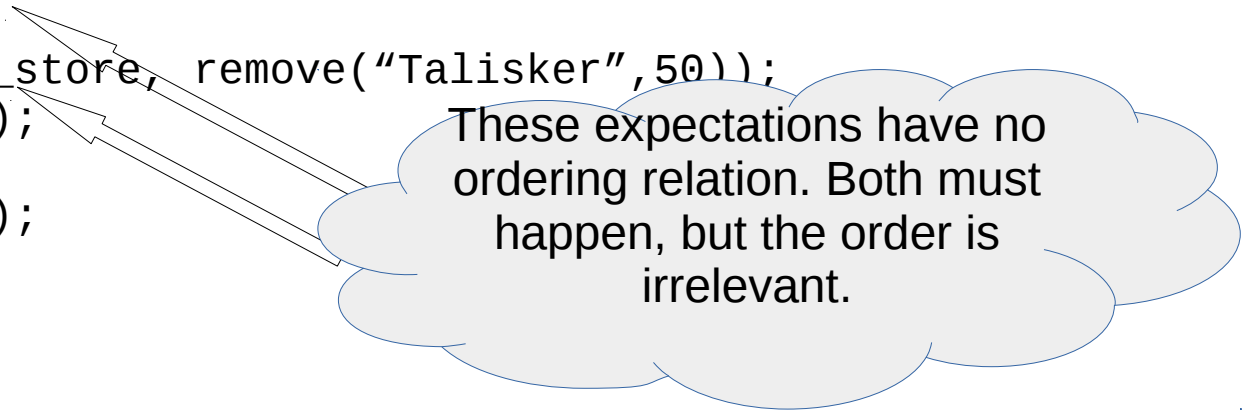
Let’s make a sloppy `order` implementation and try again:

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.remove(what, amount); s.inventory(what);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

This passes. Let’s improve the test.

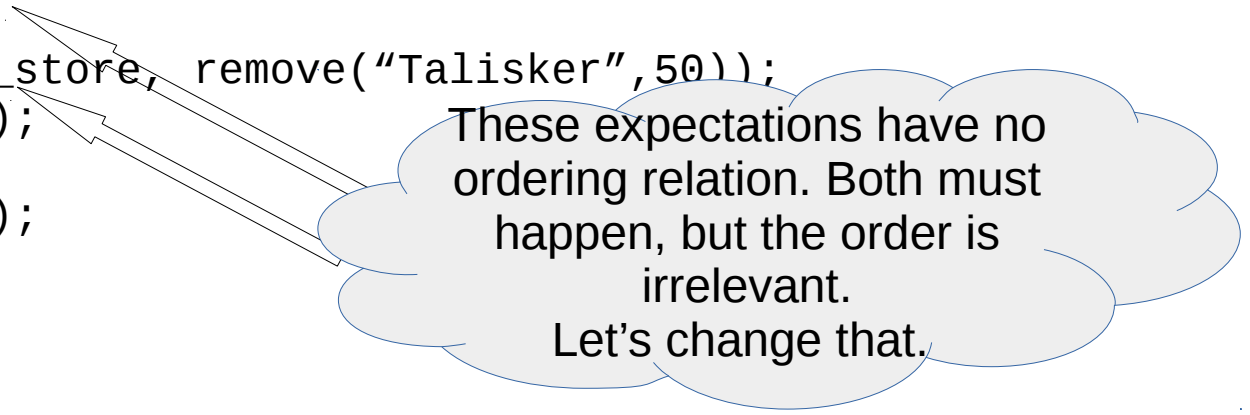
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE(“filling removes inventory when in stock”)
{
  mock_store whisky_store;
  order o{“Talisker”, 50};
  {
    REQUIRE_CALL(whisky_store, inventory(“Talisker”))
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove(“Talisker”,50));
    o.fill(whisky_store);
  }
  REQUIRE(o.is_filled());
}
```



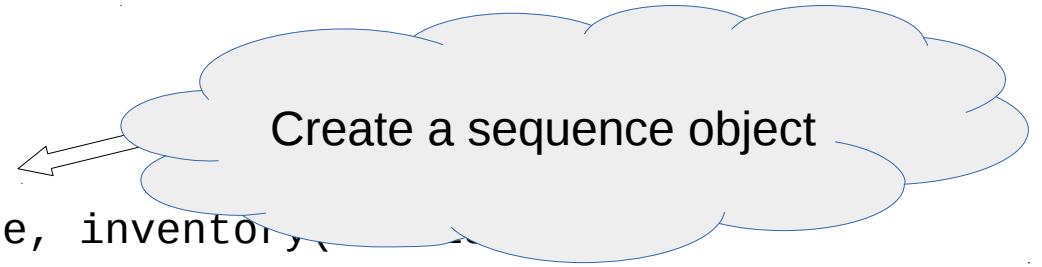
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE(“filling removes inventory when in stock”)
{
  mock_store whisky_store;
  order o{“Talisker”, 50};
  {
    REQUIRE_CALL(whisky_store, inventory(“Talisker”))
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove(“Talisker”,50));
    o.fill(whisky_store);
  }
  REQUIRE(o.is_filled());
}
```



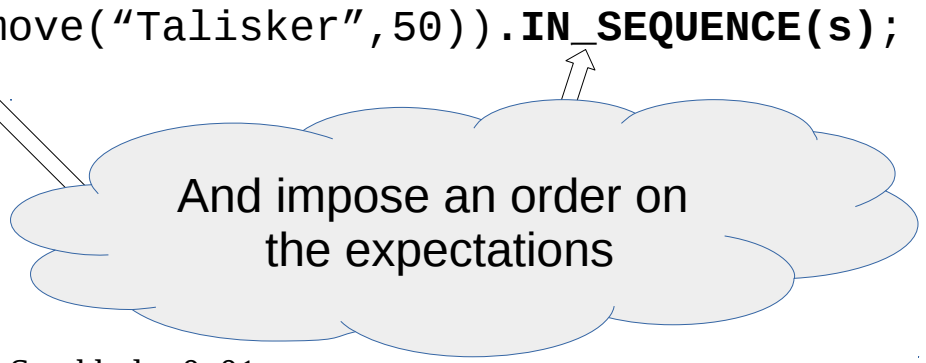
Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    trompeloeil::sequence s;
    REQUIRE_CALL(whisky_store, inventory(
      .RETURN(51);
    REQUIRE_CALL(whisky_store, remove("Talisker",50));
    o.fill(whisky_store);
  }
  REQUIRE(o.is_filled());
}
```



Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

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

TEST_CASE("filling removes inventory when in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    trompeloeil::sequence s;
    REQUIRE_CALL(whisky_store, inventory("Talisker"))
      .RETURN(51).IN_SEQUENCE(s);
    REQUIRE_CALL(whisky_store, remove("Talisker",50)).IN_SEQUENCE(s);
    o.fill(whisky_store);
  }
  REQUIRE(o.is_filled());
}
```



Mocking Modern C++ with Trompeloeil

Result from run with sloppy `order` implementation.

```
store_test.cpp:63
Sequence mismatch for sequence "s" with matching call of whisky_store.remove
("Talisker", 50) at store_test.cpp:63. Sequence "s" has whisky_store
.inventory("Talisker") at store_test.cpp:61 first in line

...
```

Mocking Modern C++ with Trompeloeil

Result from run with sloppy `order` implementation.

```
store_test.cpp:63
Sequence mismatch for sequence "s" with matching call of whisky_store.remove
("Talisker", 50) at store_test.cpp:63. Sequence "s" has whisky_store
.inventory("Talisker") at store_test.cpp:61 first in line

...
```

So `whisky_store.remove()` was called without first having called `whisky_store.inventory()`.

Mocking Modern C++ with Trompeloeil

Let’s fix the order issue

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.remove(what, amount); s.inventory(what);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

Mocking Modern C++ with Trompeloeil

Let’s fix the order issue

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.remove(what, amount); s.inventory(what);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

Mocking Modern C++ with Trompeloeil

Let’s fix the order issue

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.inventory(what); s.remove(what, amount);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

This passes. Let’s add a negative test.

Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

TEST_CASE("filling does not remove if not enough in stock")
{
  mock_store whisky_store;
  order o{"Talisker", 50};
  {
    REQUIRE_CALL(whisky_store, inventory("Talisker"))
      .RETURN(49);
    o.fill(whisky_store);
  }
  REQUIRE(!o.is_filled());
}
```

Mocking Modern C++ with Trompeloeil

```
#include <trompeloeil.hpp>

TEST_CASE(“filling does not remove if not enough in stock”)
{
  mock_store whisky_store;
  order o{“Talisker”, 50};
  {
    REQUIRE_CALL(whisky_store, inventory(“Talisker”))
      .RETURN(49);
    o.fill(whisky_store);
  }
  REQUIRE(!o.is_filled());
}
```

With only 49 in stock, nothing can be removed from the store.

And the order will not be filled.

Mocking Modern C++ with Trompeloeil

Result from run of negative test.

```
No match for call of remove with signature void(const item&, size_t) with.
  param  _1 == Talisker
  param  _2 == 50
```

Mocking Modern C++ with Trompeloeil

Result from run of negative test.

```
No match for call of remove with signature void(const item&, size_t) with.
  param  _1 == Talisker
  param  _2 == 50
```

So, it removes the items, even though there’s not enough in stock.

Mocking Modern C++ with Trompeloeil

Let’s only remove when in stock

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) { s.inventory(what); s.remove(what, amount);}
  bool is_filled() const { return true ; }
private:
  std::string what;
  size_t amount;
};
```

Mocking Modern C++ with Trompeloeil

Let’s only remove when in stock

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) {
    if (s.inventory(what) >= amount) {
      s.remove(what, amount);
      filled = true;
    }
  }
  bool is_filled() const { return filled; }
private:
  std::string what;
  size_t amount;
  bool filled = false;
};
```

Mocking Modern C++ with Trompeloeil

Let’s only remove when in stock

```
class order {
public:
  order(std::string article, size_t count)
  : what(article), amount(count) {}
  void fill(store& s) {
    if (s.inventory(what) >= amount) {
      s.remove(what, amount);
      filled = true;
    }
  }
  bool is_filled() const { return filled; }
private:
  std::string what;
  size_t amount;
  bool filled = false;
};
```

This passes!

Mocking Modern C++ with Trompeloeil

That was the basics

Mocking Modern C++ with Trompeloeil

That was the basics

Let’s begin exploring expressive power!

Mocking Modern C++ with Trompeloeil

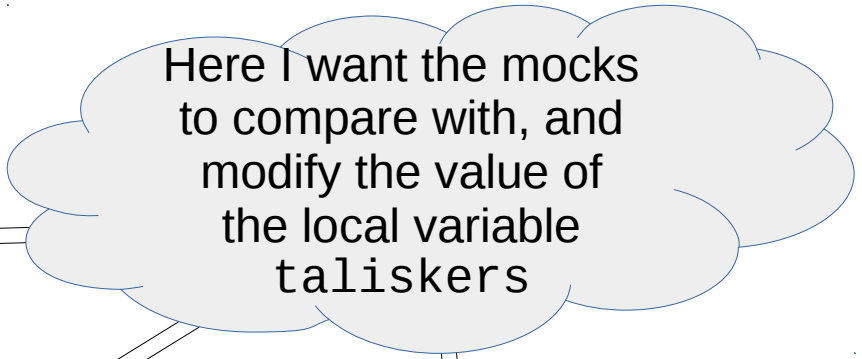
Aim for generic setup and a trivial inventory.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      order o{"Talisker", 50 };
      THEN("stock is reduced and order fulfilled ") {
        auto expectations = stock_up_store(store, taliskers);
        o.fill(store);
        REQUIRE(o.is_filled());
        REQUIRE(taliskers == 0);
      }
    }
    WHEN("order exceeds stock") {
      ...
    }
  }
}
```

Mocking Modern C++ with Trompeloeil

Aim for generic setup and a trivial inventory.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      order o{"Talisker", 50 };
      THEN("stock is reduced and order fulfilled ") {
        auto expectations = stock_up_store(store, taliskers);
        o.fill(store);
        REQUIRE(o.is_filled());
        REQUIRE(taliskers == 0);
      }
    }
    WHEN("order exceeds stock") {
      ...
    }
  }
}
```



Mocking Modern C++ with Trompeloeil

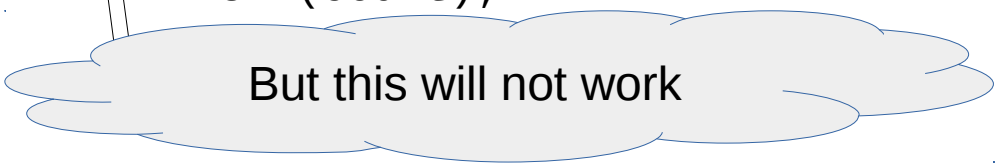
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  REQUIRE_CALL(store, inventory("Talisker"))
    .RETURN(count);









}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  REQUIRE_CALL(store, inventory("Talisker"))
    RETURN(count);
```

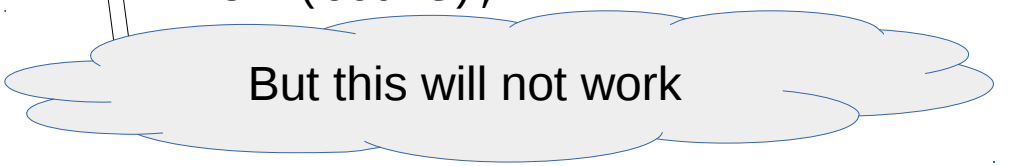


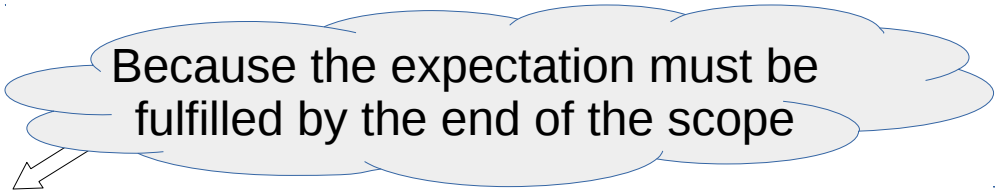

```
}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  REQUIRE_CALL(store, inventory("Talisker"))
    RETURN(count);
}
```





Mocking Modern C++ with Trompeloeil

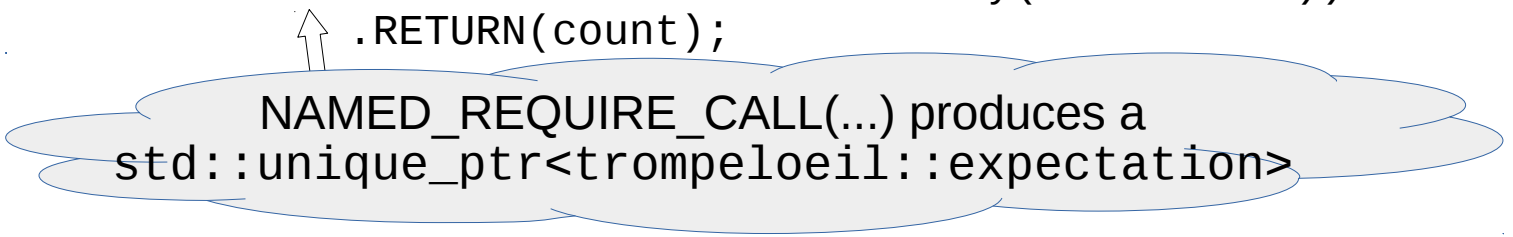
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                         .RETURN(count);









}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                       .RETURN(count);

}
```



Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                       .RETURN(count);

}
```

NAMED_REQUIRE_CALL(...) produces a std::unique_ptr<trompeloeil::expectation>

The expectation must be fulfilled by the time the object is destroyed

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                       .RETURN(count);

  return get_inventory;
}
```

NAMED_REQUIRE_CALL(...) produces a std::unique_ptr<trompeloeil::expectation>

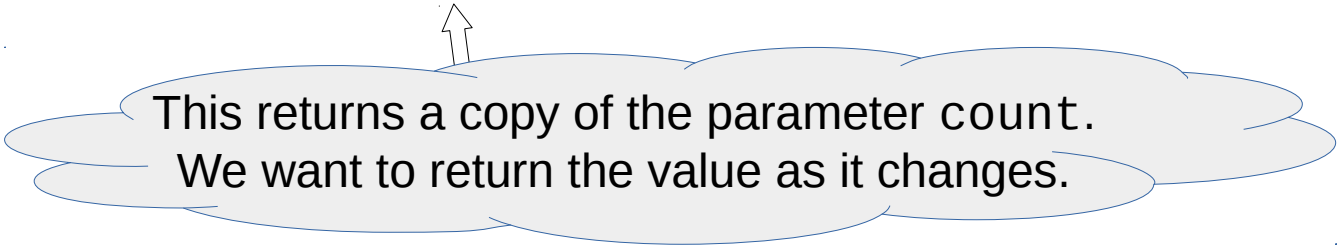
The expectation must be fulfilled by the time the object is destroyed

So we can just return it.

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                       .RETURN(count);

  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

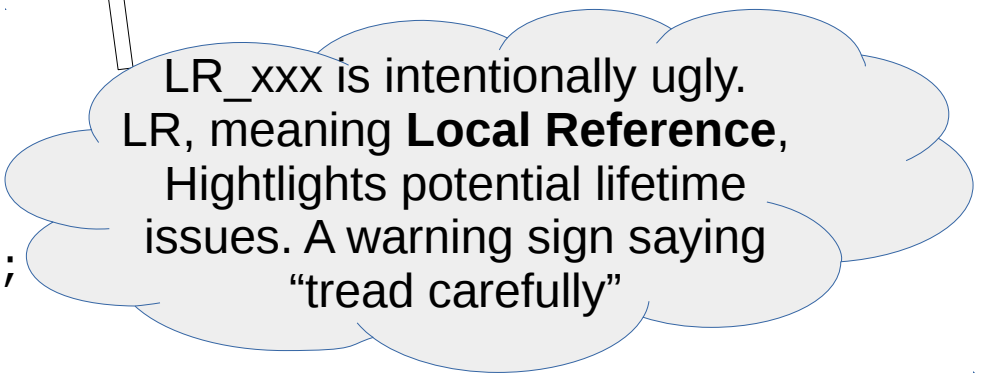
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                       .LR_RETURN(count);




  return get_inventory;
}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                       .LR_RETURN(count);

  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

Aim for generic setup and a trivial inventory.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      order o{"Talisker", 50 };
      THEN("stock is reduced and order fulfilled ") {
        auto expectations = stock_up_store(store, taliskers);
        o.fill(store);
        REQUIRE(o.is_filled());
        REQUIRE(taliskers == 0);
      }
    }
    WHEN("order exceeds stock") {
      ...
    }
  }
}
```

Mocking Modern C++ with Trompeloeil

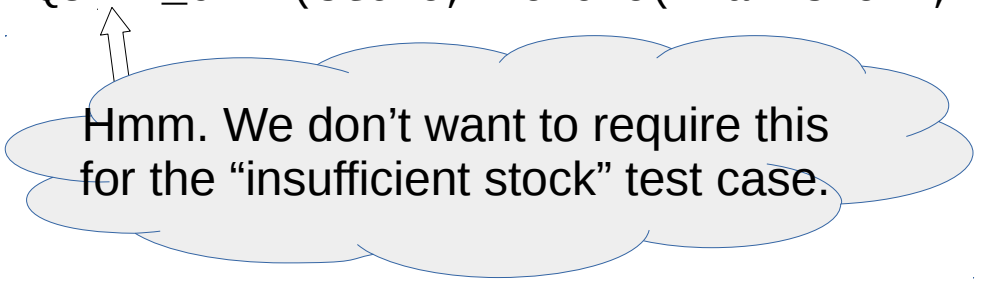
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_REQUIRE_CALL(store, remove("Talisker",…));




  return get_inventory;
}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_REQUIRE_CALL(store, remove("Talisker",…));



  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

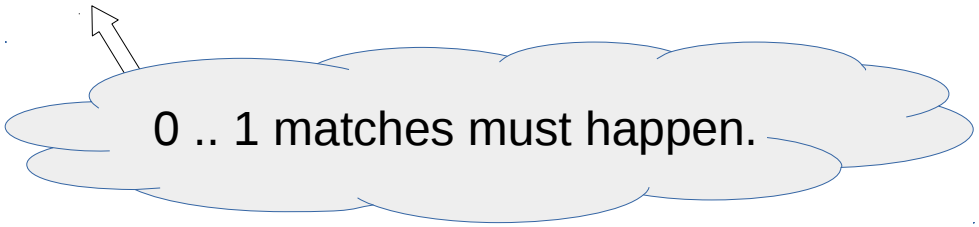
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_REQUIRE_CALL(store, remove("Talisker",…));




  return get_inventory;
}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_REQUIRE_CALL(store, remove("Talisker",…))
                    .TIMES(0,1);

  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

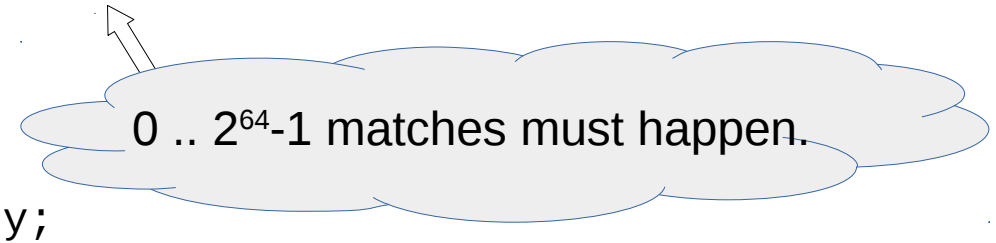
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_REQUIRE_CALL(store, remove("Talisker",…))
                    .TIMES(AT_MOST(1));




  return get_inventory;
}
```

Mocking Modern C++ with Trompeloeil

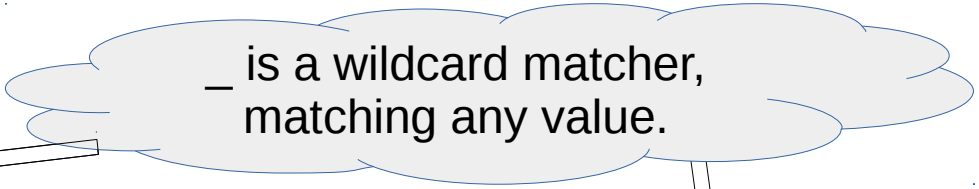
Creating a fixture with expectations.

```
#include <trompeloeil.hpp>

auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_ALLOW_CALL(store, remove("Talisker",…));

  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

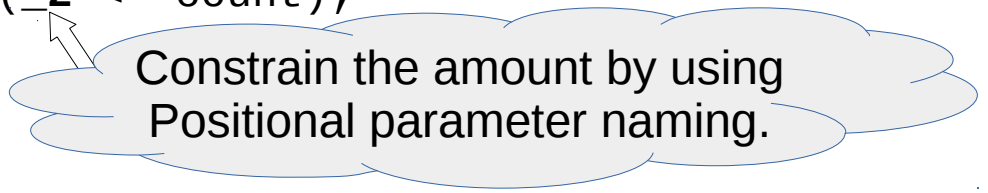



```
#include <trompeloeil.hpp>
using trompeloeil::_;
auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_ALLOW_CALL(store, remove("Talisker",_));




  return get_inventory;
}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

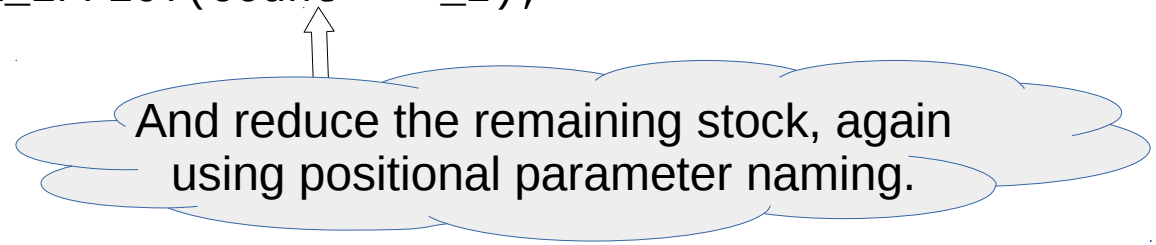
```
#include <trompeloeil.hpp>
using trompeloeil::_;
auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_ALLOW_CALL(store, remove("Talisker",_))
                    .LR_WITH(_2 <= count);

  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>
using trompeloeil::_;
auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_ALLOW_CALL(store, remove("Talisker",_))
                    .LR_WITH(_2 <= count)
                    .LR_SIDE_EFFECT(count -= _2);

  return get_inventory;
}
```



Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>
using trompeloeil::_;
auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_ALLOW_CALL(store, remove("Talisker",_))
                    .LR_WITH(_2 <= count)
                    .LR_SIDE_EFFECT(count -= _2);



  return std::make_tuple(std::move(get_inventory),
                         std::move(remove));
}
```

Mocking Modern C++ with Trompeloeil

Creating a fixture with expectations.

```
#include <trompeloeil.hpp>
using trompeloeil::_;
auto stock_up_store(mock_store& store, size_t& count)
{
  auto get_inventory = NAMED_REQUIRE_CALL(store,
                                          inventory("Talisker"))
                           .LR_RETURN(count);
  auto remove = NAMED_ALLOW_CALL(store, remove("Talisker",_))
                    .LR_WITH(_2 <= count)
                    .LR_SIDE_EFFECT(count -= _2);



  return std::make_tuple(std::move(get_inventory),
                         std::move(remove));
}
```

This works.

Mocking Modern C++ with Trompeloeil

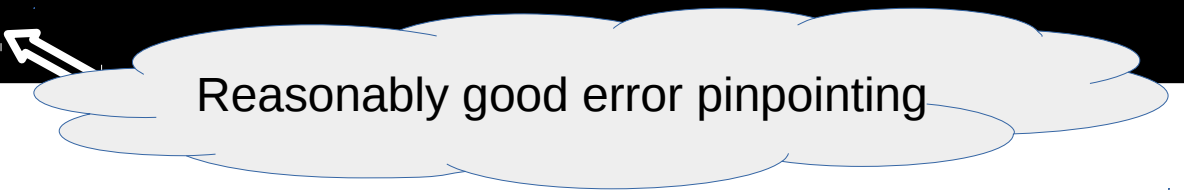
Introducing a deliberate error in the order class, making it fill more than stock, gives:

```
filling an order
     Given: a store with whiskies
      When: stock is sufficient
      Then: stock is reduced and order fulfilled
-------------------------------------------------------------------------
store_test.cpp:95
.........................................................................

store_test.cpp:19: FAILED:
explicitly with message:
  No match for call of remove with signature void(const item&, size_t) with.
    param  _1 == Talisker
    param  _2 == 51

  Tried store.remove("Talisker",_) at store_test.cpp:88
    Failed WITH(_2 <= count)
```

Mocking Modern C++ with Trompeloeil

Introducing a deliberate error in the order class, making it fill more than stock, gives:

```
filling an order
     Given: a store with whiskies
      When: stock is sufficient
      Then: stock is reduced and order fulfilled
-------------------------------------------------------------------------
store_test.cpp:95
.........................................................................

store_test.cpp:19: FAILED:
explicitly with message:
  No match for call of remove with signature void(const item&, size_t) with.
    param  _1 == Talisker
    param  _2 == 51

  Tried store.remove("Talisker",_) at store_test.cpp:88
    Failed WITH(_2 <= count)
```



Mocking Modern C++ with Trompeloeil

Let's look at the "**order exceeds stock**" case.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      ...
    }
    WHEN("order exceeds stock") {
      order o{"Talisker", 51 };
      THEN("stock is untouched and order not fulfilled") {
        auto expectations = stock_up_store(store, taliskers);

        o.fill(store);
        REQUIRE(!o.is_filled());
        REQUIRE(taliskers == 50);
      }
    }
  }
}
```

Mocking Modern C++ with Trompeloeil

Let's look at the "order exceeds stock" case.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      ...
    }
    WHEN("order exceeds stock") {
      order o{"Talisker", 51 };
      THEN("stock is untouched and order not fulfilled") {
        auto expectations = stock_up_store(store, taliskers);

        o.fill(store);
        REQUIRE(!o.is_filled());
        REQUIRE(taliskers == 50);
      }
    }
  }
}
```

We check that the stock is not touched, but there's no preventing that `fill(store)` calls `remove("Talisker", 0)`

Mocking Modern C++ with Trompeloeil

Let’s look at the order exceeds stock case.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      ...
    }
    WHEN("order exceeds stock") {
      order o{"Talisker", 51 };
      THEN("stock is untouched and order not fulfilled") {
        auto expectations = stock_up_store(store, taliskers);
        REQUIRE_CALL(store, remove(_,_)).TIMES(0);
        o.fill(store);
        REQUIRE(!o.is_filled());
        REQUIRE(taliskers == 50);
      }
    }
  }
}
```

Mocking Modern C++ with Trompeloeil

Let’s look at the order exceeds stock case.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      ...
    }
    WHEN("order exceeds stock") {
      order o{"Talisker", 51 };
      THEN("stock is untouched and order not fulfilled") {
        auto expectations = stock_up_store(store, taliskers);
        FORBID_CALL(store, remove(_,_));
        o.fill(store);
        REQUIRE(!o.is_filled());
        REQUIRE(taliskers == 50);
      }
    }
  }
}
```

Mocking Modern C++ with Trompeloeil

Let's look at the order exceeds stock case.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      ...
    }
    WHEN("order exceeds stock") {
      order o{"Talisker", 51 };
      THEN("stock is untouched and order not fulfilled") {
        auto expectations = stock_up_store(store, taliskers);
        FORBID_CALL(store, remove(_,_));
        o.fill(store);
        REQUIRE(!o.is_filled());
        REQUIRE(taliskers == 50);
      }
    }
  }
}
```

Now we have two conflicting expectations for `remove()`.
`ALLOW` from `stock_up_store()`, and `FORBID`.

Mocking Modern C++ with Trompeloeil

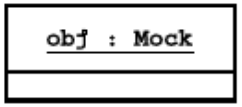
Let's look at the order exceeds stock case.

```
#include <trompeloeil.hpp>

TEST_CASE("filling an order") {
  GIVEN("a store with whiskies") {
    mock_store store;
    size_t taliskers = 50;
    WHEN("stock is sufficient" ) {
      ...
    }
    WHEN("order exceeds stock") {
      order o{"Talisker", 51 };
      THEN("stock is untouched and order not fulfilled") {
        auto expectations = stock_up_store(store, taliskers);
        FORBID_CALL(store, remove(_,_));
        o.fill(store);
        ...
      }
    }
  }
}
```

When there are multiple active expectations, they are matched in reversed order of creation, so that you can state generous defaults and local constraints.

Now we have two conflicting expectations for `remove()`. `ALLOW` from `stock_up_store()`, and `FORBID`.

Mocking Modern C++ with Trompeloeil



Ceci n'est pas un obj ;

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

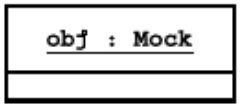
`.IN_SEQUENCE(s...)` ← Impose an ordering relation between expectations by using **sequence** objects

`.TIMES(min {, max} )` ← 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)`

Mocking Modern C++ with Trompeloeil



Trompeloeil cheat sheet for matchers and object life time management.

Ceci n'est pas un obj ;

Matchers. Substitute for values in parameter list of expectations.

Any type allowing op				*Disambiguated type*
`_`	any value			`ANY(type)`
`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`**

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

Mocking Modern C++ with Trompeloeil

Björn Fahller

 bjorn@fahller.se

 @bjorn_fahller

Mocking Modern C++ with Trompeloeil

https://github.com/rollbear/trompeloeil

Ceci n'est pas un objet

Björn Fahller

 bjorn@fahller.se

 @bjorn_fahller