Backporting to the Future

(in 20 minutes)

Andrew Paxie

26 September 2018

"Ex Ignorantia Ad Sapientiam; Ex Luce Ad Tenebras"

Contents

Contents

Trompeloeil

A header only C++ mocking framework

https://github.com/rollbear/trompeloeil

Björn Fahller
(Code Owner)

mailto:bjorn@fahller.se

https://github.com/rollbear

https://playfulprogramming.blogspot.com

Andrew Paxie
(Contributor)

mailto:cpp.scribe@gmail.com

https://github.com/AndrewPaxie

https://blog.andrew.paxie.org

Trompeloeil

Main features:

- Mock functions
- Expectations
- Modifiers
- Matchers

Other features:

- Sequencing expectations
- Object-lifetime monitoring
- Integration into test framework reporting
- Tracing

Trompeloeil

Mock functions

- `MAKE_MOCKn(name, sig{, spec})`
- `MAKE_CONST_MOCKn(name, sig{, spec})`

```
struct Interface
{
  virtual void setValue(int v) = 0;
  virtual int getValue() const = 0;
};

struct Mock: Interface
{
  MAKE_MOCK1(setValue, void(int), override);
  MAKE_CONST_MOCK0(getValue, int(), override);
};
```

Trompeloeil

Expectations

- `REQUIRE_CALL(obj, func(params))`
- `ALLOW_CALL(obj, func(params))`
- `FORBID_CALL(obj, func(params))`

```
TEST_CASE("Unit test", "[Sample]")
{
// Setup
  Mock obj;
  REQUIRE_CALL(obj, setValue(ANY(int));
  FORBID_CALL(obj, getValue());
}
```

Also: Named variants of the above

Trompeloeil

Modifiers

- `WITH(condition)`
- `SIDE_EFFECT(statement)`
- `RETURN(expression)`
- `THROW(expression)`

Parameters named using `_1 ... _15` placeholder variables

Also: local reference (`LR_`) versions of the above

Trompeloeil

Matchers

`_`	`ANY(type)`
`eq(mark)`	`ne(mark)`
`ge(mark)`	`le(mark)`
`gt(mark)`	`lt(mark)`
`re(mark, ...)`	
`!`	negate matcher
`*`	pointer dereference

Trompeloeil

Example

```
// Production class
struct Db: IDb
{
  virtual
  int
  lookup(const char*);
};

// SUT
struct Engine
{
  explicit
  Engine(IDb& db)
    : db(db_)
  {}

  int
  compute(
    const char* key)
  {
    return 3 * db.lookup(key);
  }

private:
  IDb& db;
}
```

```
// Interface
struct IDb
{
  virtual
  int
  lookup(const char*) = 0;
};

// Mock class
struct MockDb final: IDb
{
  // Mock function
  MAKE_MOCK1(
    lookup,
    int(const char*),
    final);
};
```

```
// Unit test
TEST_CASE(
  "Compute, key exists",
  "[Engine]")
{
// Setup
  const char* key = "foo";

  MockDb db;

  REQUIRE_CALL(db, lookup(key))
    .RETURN(2);

  Engine sut(db);

// Exercise
  int ret = sut.compute(key);

// Verify
  REQUIRE(ret == 6);

// Teardown
  // Final verify in destructor
}
```

Contents

Goals

- Provide existing API in C++11 mode
- Support same compilers, versions
- Preserve C++14 capability

Contents

Unused features: Library (1)

`constexpr` for `<complex>`	[N3302]
Making operator functors `greater<>`	[N3421]
`std::result_of` and SFINAE	[N3462]
`constexpr` for `<chrono>`	[N3469]
`constexpr` for `<array>`	[N3470]
Improved `std::integral_constant`	[N3545]

Unused features: Library (2)

Null forward iterators	[N3644]
`std::quoted`	[N3654]
Heterogeneous associative lookup	[N3657]
Shared locking in C++	[N3659]
Fixing `constexpr` member functions without `const`	[N3669]
`std::get<T>`	[N3670]

Unused features: Language

Binary literals	[N3472]
Variable templates	[N3651]
Extended `constexpr`	[N3652]
Member initializers and aggregates	[N3653]
`[[deprecated]]` attribute	[N3760]
Single quote as digit separator	[N3781]

Unused features: Miscellaneous

Clarifying memory allocation	[N3664]
Sized deallocation	[N3778]

Used features - Library

`constexpr` for `<initializer_list>`, `<utility>` and `<tuple>`	[N3471]
User-defined literals for `<chrono>` and `<string>`	[N3642]
TransformationTraits Redux, v2	[N3655]
`std::make_unique`	[N3656]
`std::integer_sequence`	[N3658]
`std::exchange`	[N3668]
Dual-range `std::equal`, `std::is_permutation`, `std::mismatch`	[N3671]

Used features - Library

`constexpr` for `<initializer_list>`, `<utility>` and `<tuple>`	[N3471]
User-defined literals for `<chrono>` and `<string>`	[N3642]
TransformationTraits Redux, v2	[N3655]
`std::make_unique`	[N3656]
`std::integer_sequence`	[N3658]
`std::exchange`	[N3668]
Dual-range `std::equal`, `std::is_permutation`, `std::mismatch`	[N3671]

Used features - Language

Tweak to certain contextual conversions	[N3323]
`decltype(auto)` and return type deduction for normal functions	[N3638]
Generalized lambda captures	[N3648]
Generic lambda expressions	[N3649]

Contents

Backporting: Library
Approach

- Define a `namespace detail`
  - Define C++11 versions of the C++14 API.
- Call the `namespace detail` entities.
  - `std::make_unique` becomes `detail::make_unique`.
- For C++14 and later, make `std::` entities accessible in namespace `detail`.
  - Maybe a namespace alias: `namespace detail = std;`
  - Maybe using declarations in `namespace detail`
  - Maybe alias templates in `namespace detail`

Backporting: Library

Affected C++ Standard Library headers

- `<memory>`
- `<type_traits>`
- `<utility>`

Backporting: Library

`<memory>`

- `make_unique`

Thanks:
Stephan T. Lavavej [N3656]

Backporting: Library

<type_traits>

- conditional_t
- decay_t
- enable_if_t
- remove_pointer_t
- remove_reference_t

Thanks:
Walter E. Brown [N3655]

Backporting: Library

`<utility>`

- `exchange`
- `integer_sequence`
- `index_sequence`
- `make_integer_sequence`
- `make_index_sequence`
- `index_sequence_for`

Thanks:

Jeffrey Yasskin	[N3688]
Jonathan Wakely	[N3658]
Peter Dimov	[Boost.mp11]

Contents

Backporting: Language

- Tweak to certain contextual conversions
- Generic lambda expressions
- Generalized lambda captures
- Return type deduction for normal functions
- `decltype(auto)`

Generic lambda expressions
Definition

- Lambdas that use `auto` in their parameter specifications
- In C++14, lambdas may also be variadic

```
[](auto x)
{
  return x + x;
}
```

```
[](auto&&... xs)
{
  return sum(
    std::forward<
      decltype(xs)
    >(xs)...);
}
```

Replace generic lambdas
Example

```
[](auto x)
{
  return x + x;
}
```

```
class ClosureType
{
public:
  template <typename T>
  auto
  operator()(T x) const
  {
    return x + x;
  }
};
```

Replace generic lambdas
Approach

- Replace generic lambda with lambda
- Use a *functor* with function call operator member template (*generic functor*)
- Replace functions returning generic lambda with generic functor
- Simulate *init-capture*s using constructor and member variables

Generalized lambda capture

Definition

An *init-capture* may specify

- A name of the data member in the closure type
- An expression to initialize that data member

Useful for capturing

- A move-only object
- An object that's expensive to copy but cheap to move

Replace generalized lambda capture

Example

```
auto p =
  std::make_unique<
    std::vector<int>>();

[ptr = std::move(p)]
{
  return ptr->empty();
}
```

```
class ClosureType
{
  using T =
    std::unique_ptr<std::vector<int>>;

public:
  explicit ClosureType(T&& p)
    : ptr(std::move(p))
  {}

  bool operator()() const
  {
    return ptr->empty();
  }

private:
  T ptr;
};
```

Replace generalized lambda capture

Approaches

Create a functor like `ClosureType`

- Declare member variables
- Define constructor to initialize members

Use `std::bind` [Meyers, Item 32]

- Move object to be captured into a function object produced by `std::bind`
- Give the lambda a reference to the captured object

Return type deduction

Definition

Merrill, [N3638]:

> *Write `auto` on your function declaration and have the return type deduced*

```
auto
foo(int var)
{
  if (var)
  {
    return 0;
  }
  else
  {
    return var + 1;
  }
}
```

Replace return type deduction

Approach

Use trailing return type

```
auto
foo(int var)

{
  if (var)
  {
    return 0;
  }
  else
  {
    return var + 1;
  }
}
```

```
auto
foo(int var)
-> int
{
  if (var)
  {
    return 0;
  }
  else
  {
    return var + 1;
  }
}
```

`decltype(auto)`

Definition

Use the rules of `decltype()` to deduce a type.

Merrill, [N3638]:

> *Plain `auto` never deduces to a reference, and `auto&&` always deduces to a reference. [...] forwarding functions can't use `auto`.*

Replace `decltype(auto)`

Approach

- Replace `decltype(auto)` with explicit type
- Use `auto` and trailing return type

Replace decltype(auto)

Example: Use explicit type

From macro TROMPELOEIL_RETURN_(...):

```
[&](auto& trompeloeil_x)
-> decltype(auto)
{
  // Define placeholders
  // from trompeloeil_x

  return __VA_ARGS__;
}
```

```
[&](auto& trompeloeil_x)
-> trompeloeil_return_of_t
{
  // Define placeholders
  // from trompeloeil_x

  return __VA_ARGS__;
}
```

Replace decltype(auto)

Example: Use auto and trailing return type

From placeholder naming code (simplified):

```
template <
  int N,
  typename T
>
constexpr
decltype(auto)
arg(
  T* t,
  std::true_type)

{
  return std::get<N-1>(*t);
}
```

```
template <
  int N,
  typename T
>
constexpr
auto
arg(
  T* t,
  std::true_type)
-> decltype(std::get<N-1>(*t))
{
  return std::get<N-1>(*t);
}
```

Contents

Goals reviewed

- Provide existing API in C++11 mode
- Support same compilers, versions
- Preserve C++14 capability

Trompeloeil

https://github.com/rollbear/trompeloeil

Björn Fahller
(Code Owner)

mailto:bjorn@fahller.se

https://github.com/rollbear

https://playfulprogramming.blogspot.com

Andrew Paxie
(Contributor)

mailto:cpp.scribe@gmail.com

https://github.com/AndrewPaxie

https://blog.andrew.paxie.org