Awaiting for the ranges

Meeting C++, Berlin 2015

Ivan Čukić

ivan.cukic@kde.org
http://cukic.co

About me

- KDE development
- Talks and teaching
- Functional programming enthusiast

Meeting C++ 2014

Soviet War Memorial
Treptower Park
Berlin

C++ Russia 2015

Park Pobedy
Metro Station
Moscow

Disclaimer

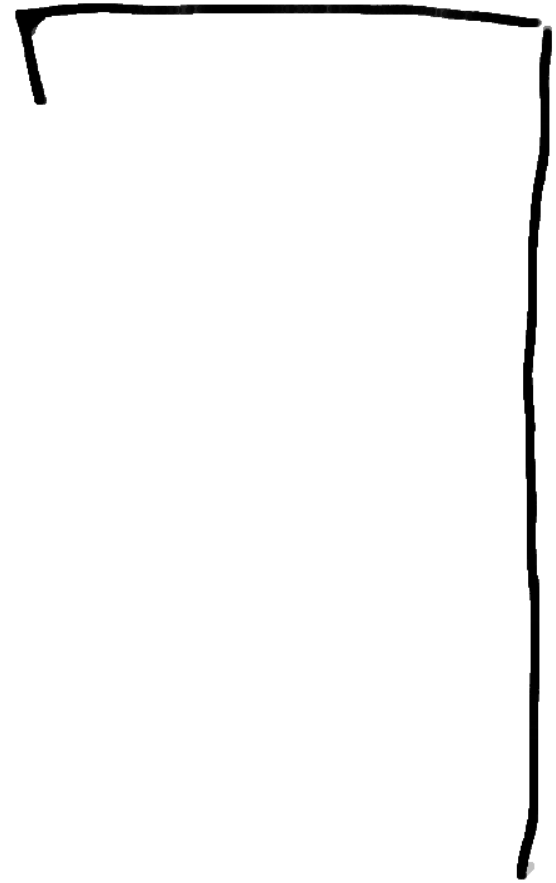
> Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.
>
> Philip Wadler

Disclaimer

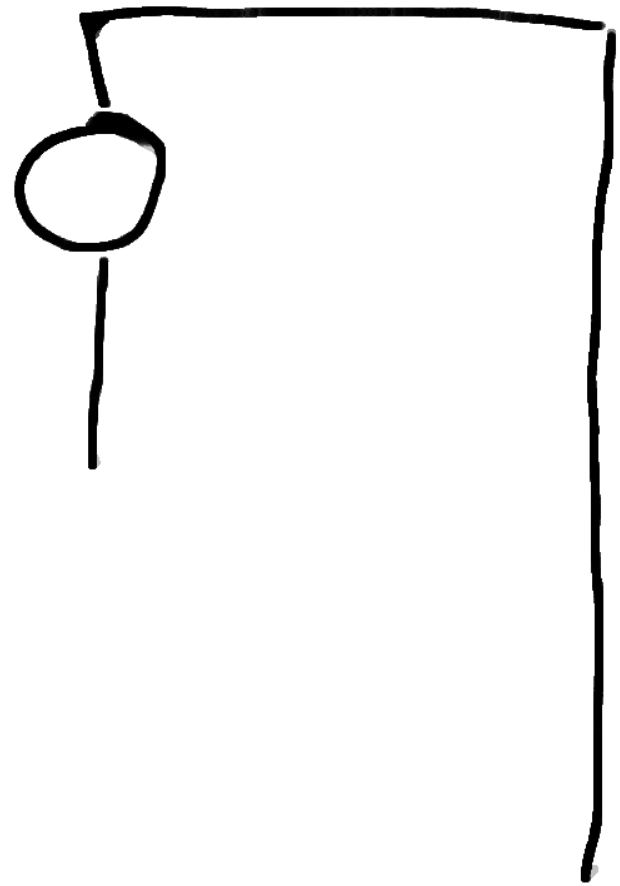
The code snippets are optimized for presentation, it is not production-ready code.

`std` namespace is omitted, value arguments used instead of const-refs or forwarding refs, etc.

AWAITING
FOR
THE
RANGES



AWAITING FOR THE ~~RANGES~~

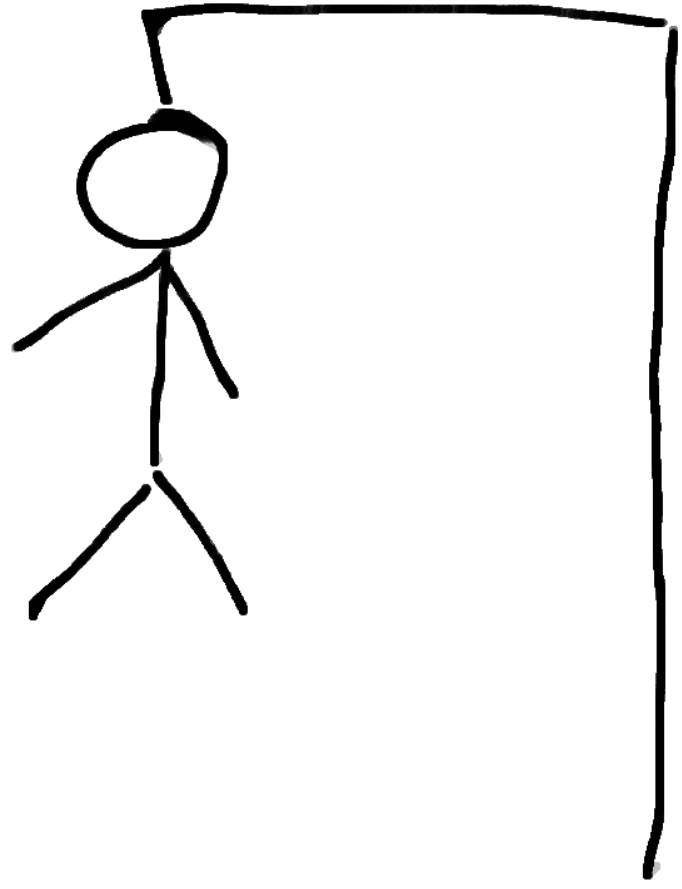


AWAITING

FOR

THE

RANGES

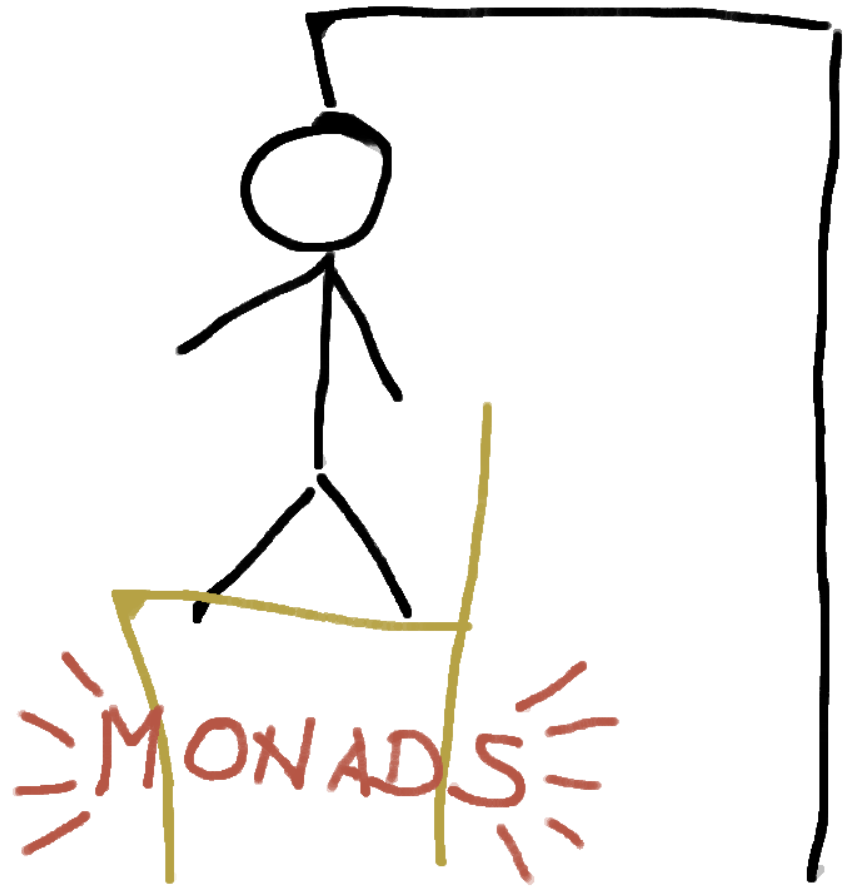


AWAITING

FOR

THE

RANGES



RANGES

Almost like Nicola's example

```
vector<int> xs = {1, 2, 3, 4, 5, ...};

for (auto &x: xs) {
    x = x * x;
}

// xs = {1, 4, 9, 16, 25}
```

Almost like Nicola's example

```
vector<int> xs = {1, 2, 3, 4, 5, ...};
int sum = 0;

for (auto &x: xs) {
    sum += x * x;
}

// xs = {1, 4, 9, 16, 25}
```

Haskell

There are projects where annotations like
`could_have_been_one_line_in_haskell`
are a common thing

```
sum $ map (λ x → x * x) xs
```

Item 1: Consider using STL algorithms

```
vector<int> xs = {1, 2, 3, 4, 5, ...};
vector<int> squares;

std::transform(xs.cbegin(), xs.cend(),
               std::back_inserter(squares),
               [] (int x) { return x * x; });

int result = std::accumulate(squares.cbegin(),
                             squares.cend());
```

Item 1: Consider using STL algorithms, but not at all cost

```
vector<int> xs = {1, 2, 3, 4, 5, ...};
vector<int> squares(xs.size());

std::transform(xs.cbegin(), xs.cend(),
               squares.begin(),
               [] (int x) { return x * x; });

int result = std::accumulate(squares.cbegin(),
                             squares.cend());
```

Item 2: Consider using ranges

There are projects where annotations like `could_have_been_one_line_in_cxx` should become a common thing

```
accumulate(xs | transform(
                    [] (auto x) {
                        return x * x;
                    }
                ));

accumulate(xs | transform(_1 * _1));
                    // or _ * _
                    // or arg1 * arg2
```

Item 2: Consider using ranges

Some call it STL 2.o, provides separate views and actions

Filter a container using a predicate and transform it.

```
vector<int> xs {1, 2, 3, 4, 5, ...};
auto rng = xs | view::filter([] (int i) { return i % 2 == 1; })
              | view::transform([] (int i) { return to_string(i); });
```

Generate an infinite list of integers starting at 1, square them, take the first 10, and sum them:

```
int sum = accumulate(view::ints(1)
                   | view::transform([] (int i) { return i * i; })
                   | view::take(10), 0);
```

Generate a sequence on the fly with a range comprehension and initialize a vector with it:

```
vector<int> xs =
    view::for_each(view::ints(1,10), [](int i){
        return yield_from(view::repeat(i,i));
    });
```

Item 2: Consider using ranges

```
// ys - received values
// es - expected values

sqrt(accumulate(
    zip(ys, es)
        | transform(_1 - _2)
        | transform(_1 * _1)
));
```

Item 2: Consider using ranges

```
auto mistery_method(vector<gadget> gadgets, int offset, int count)
{
    vector<gadget> result;
    int skipped = 0, took = 0;

    for (const auto &gadget: gadgets) {
        if (is_container(gadget))
            continue;

        vector<gadget> children;

        for (const auto &child: children(gadget)) {
            if (is_visible(child)) {
                if (skipped < offset) {
                    skipped++;
                } else if (took <= count) {
                    took++;
                    children.push_back(child);
                }
            }
        }

        copy(children.cbegin(), children.cend(), back_inserter(result));
    }

    return result;
}
```

Item 2: Consider using ranges

```
range<To> transform(range<From> l,
                    function<To(From)> f);

range<T> filter(range<T> l,
                function<bool(T)> predicate);

range<T> flatten(range<range<T>> lists);

range<T> take(range<T> l, int number);

range<T> drop(range<T> l, int number);
```

Item 2: Consider using ranges

```
auto mistery_method(vector<gadget> gadgets,
                    int offset, int count)
{
    return gadgets | filtered(is_container)
                   | transformed(children)
                   | flatten()
                   | filtered(is_visible)
                   | drop(offset)
                   | take(count);
}
```

Item 2: Consider using ranges

```
auto fizzs = repeat(
       ints(1, 3) |         // 1 2 3 1 2 3 1 2 3 1 2 3 ...
       transform(_1 == 3)   // 0 0 1 0 0 1 0 0 1 0 0 1 ...
    );

auto buzzs = repeat(
       ints(1, 5) |         // 1 2 3 4 5 1 2 3 4 5 1 2 ...
       transform(_1 == 5)   // 0 0 0 0 1 0 0 0 0 1 0 0 ...
    );

auto fizzbuzzs =
       zip(ints(1), fizzs, buzzs) |
                            // 100 200 310 400 501 ...
       transform([] (auto item) {
           bool fizz = get<1>(item);
           bool buzz = get<2>(item);
           return fizz && buzz ? "FizzBuzz"
                : fizz         ? "Fizz"
                : buzz         ? "Buzz"
                : /*otherwise*/  to_string(get<0>(item));
       });
```

Item 3: Consider using lambdas, but not at all cost

Lambdas are good, but sometimes cumbersome.

```
responses | filter([] (auto response) {
                        return response.error();
                    });

responses | filter(_1.error() == true); ???
```

Item 4: Consider writing functions* manually

Lambdas are good, but sometimes cumbersome.

```
class error_test_t {
public:
    error_test_t(bool error = true)
        : m_error(error)
    {}

    error_test_t operator==(bool error) const
    {
        return error_test_t(!!m_error ^ !!error));
    }

    template<typename T>
    bool operator() (T value) const
    {
        // Can be made really evil and make it test
        // for is_error(), error(), is_error, error
        return m_error == (bool)value.error();
    }

private:
    bool m_error;
};

error_test_t error(true);
error_test_t not_error(false);
```

Item 4: Consider writing functions* manually

Lambdas are good, but sometimes cumbersome.

```
responses | filter(error);
responses | filter(not_error);
responses | filter(error == true);
responses | filter(error == false);
```

Item 5: Not really an item

```
xs = { 1, 5, 7, 3 }
xs | sort | take(5)

// Should we be lazy?
xs = { 1, 5, 7, 3, 10, 42, 32, 2, ... }
xs | sort | take(5)

// Should we be eager?
xs = { 1, 5, 7, 3, 10, 42, 32, 2, ... }
accumulate(xs | sort)
```

AWAIT

Futures

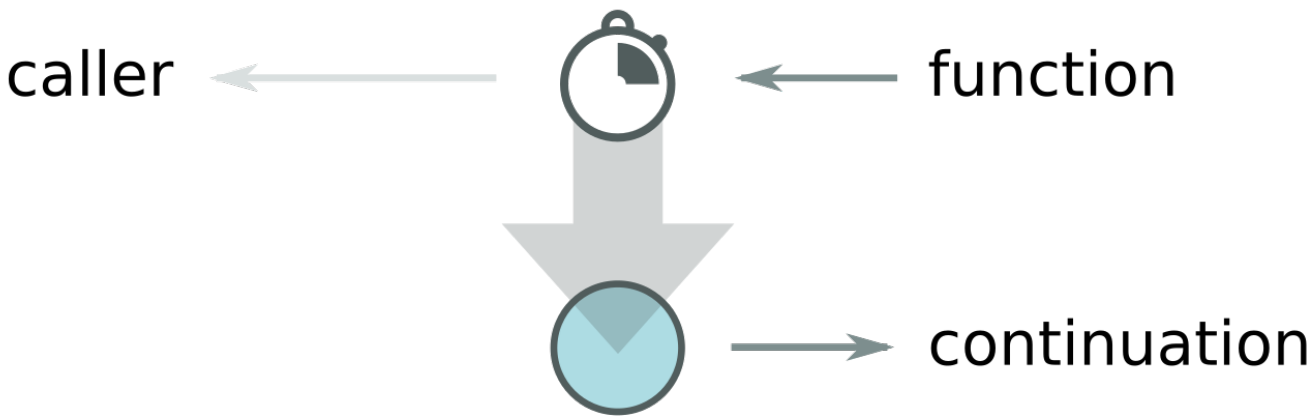
```
void get_page()
{
    get("meetingcpp.com",
        on_page_retrieved);
}

void on_page_retrieved(page_t page)
{
    for (image: page.images()) {
        image.get(on_image_retrieved);
    }
}

void on_image_retrieved(image_t image)
{ ... }
```

Real item 5: Avoid `future<T>::get()`

```
future<answer_t> answer = meaning_of_life();
// would actually be good for the mice
answer.get();
```



Futures

```
get("meetingcpp.com")
.then(
    [] (auto page) {
        for (image: page.images()) {
            image.get()
            .then(
                [] (auto image_result) {
                    ...
                }
            );
        }
    }
)
```

Futures

```
get("meetingcpp.com",
    [] (auto page) {
        for (image: page.images()) {
            image.get(
                [] (auto image_result) {
                    ...
                }
            );
        }
    }
)
```

Futures

```
auto f = get("meetingcpp.com")

...

f_somewhere_else.then(...)
```

Item ILC: Put futures in chains

```
// .then(f): future<From> → future<To>

auto f =
    get("meetingcpp.com")    // future<page_t>
    .then( ... → string ...) // future<string>
    .then( ... → int ...)    // future<int>
    ...
    .then( ... → SomeT ...)  // future<SomeT>

all_of, any_of, etc.
```

Item ILC+1: Don't wait, await

```
// get(string) → future<page_t>

page_t page =
    co_await get("meetingcpp.com");

// do something with the page object
...
```

Item ILC+1: Don't wait, await

```
// get(string) → future<page_t>

page_t page =
    co_await get("meetingcpp.com");

for (auto image: page.images()) {
    auto image = co_await image.get();

    // do something with the image
    ...
}
```

Item ILC+1: Don't wait, await

`co_await expression` is equivalent to:

```
{
    auto && tmp = <expr>;
    if (!await_ready(tmp)) {
        await_suspend(tmp, continuation);

    }
    return await_resume(tmp);
}
```

Item ILC+1: Don't wait, await

```
bool await_ready(future<T> &future) {
    return future.is_ready();
}

// or bool, if suspending can fail
void await_suspend(future<T> &future,
                   Cont trigger_continuation) {
    future.then([=](auto&) {
        trigger_continuation();
    }
}

auto await_resume(future<T> &future) {
    return future.get();
}
```

Item ILC+1: Don't wait, await

```
// get(string) → future<page_t>

??? get_meeting_cpp_page()
{
    page = co_await get("meetingcpp.com");

    // page_t::header() → header_t
    co_return page.header();
}
```

Item ILC+2: Don't abuse await

```
// create_gadget() → gadget*

void get()
{
    auto gadget = co_await create_gadget();

    // instead of
    auto gadget_ptr = create_gadget();
    if (!gadget_ptr) return;

    auto gadget = *gadget;


    ...
}
```

Item ILC+3: Optionally abuse await

```
// create_gadget() → optional<gadget>

optional<some_type> get()
{
    auto gadget = co_await create_gadget();

    // instead of
    auto gadget_opt = create_gadget();
    if (!gadget_opt.is_initialized()) return;

    auto gadget = gadget.get();


    ...


    co_return something;
}
```

Item ILC+4: Abuse await to look as cool as Gor

Deep recursion:

```
co_return co_await (fib(n - 1) + fib(n - 2));
```

See Gor Nishanov's talks

SAME OL’ SAME OL’

Two sides of the same coin?

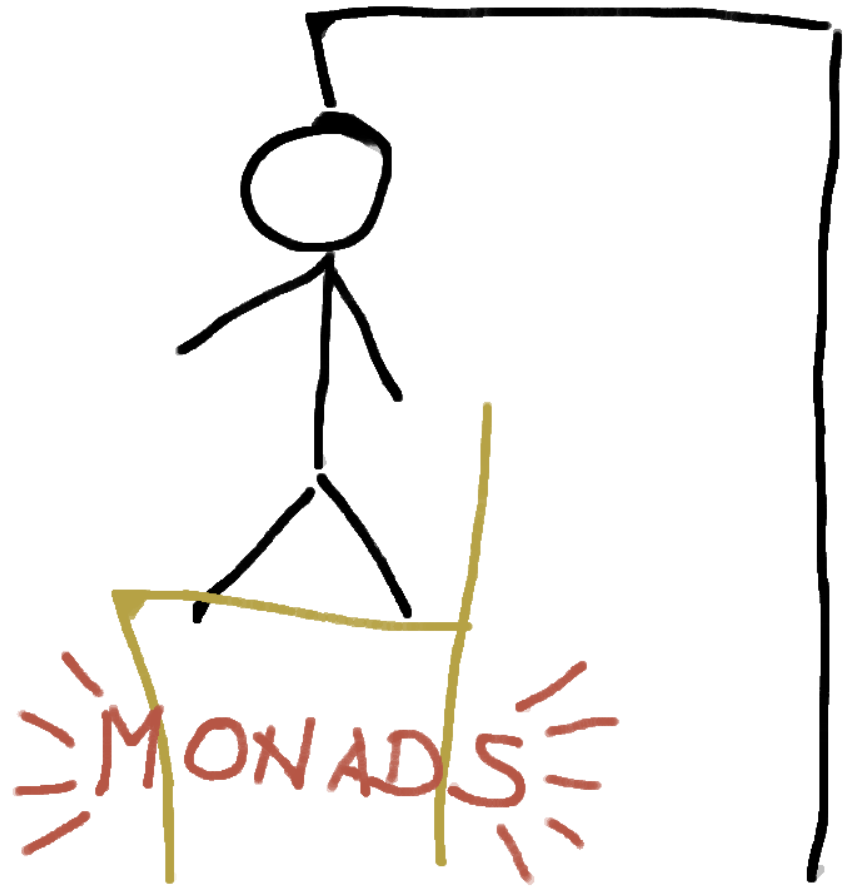
What is the connection between ranges and await?

AWAITING

FOR

THE

RANGES



Remember monads?

A container-like structure with a few methods defined on it.

- Constructor method that returns a container containing it

  `T → C<T>`

- Transform (map) method

  `(C<From>, function<From → To>) → C<To>`

- Flatten method

  `C<C<T>> → C<T>`

Short break

Which is the clearest:

1. Using the `std::function` syntax:
   ```
   (C<From>, function<To(From)>) → C<To>
   ```
2. Using arrows in `function`
   ```
   (C<From>, function<From → To>) → C<To>
   ```
3. Not writing `function` at all
   ```
   (C<From>, From → To) → C<To>
   ```

Monads as containers

Constructor method that returns a container containing it

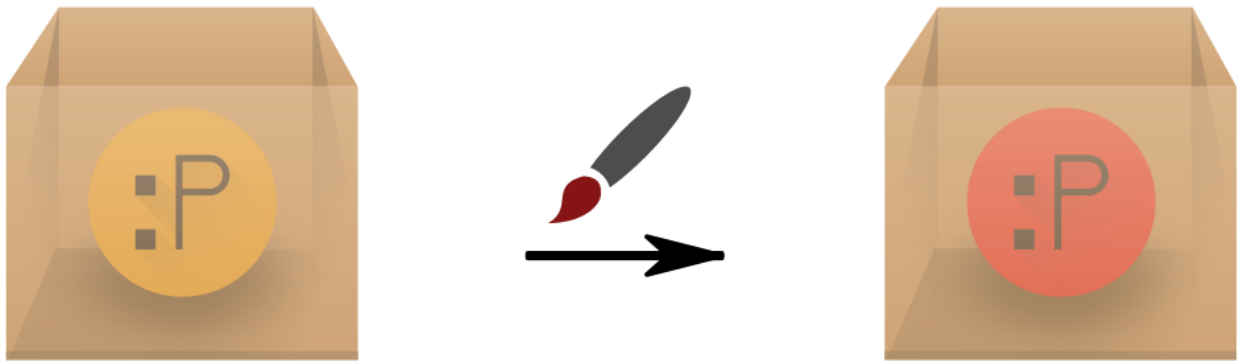
`(T) -> C<T>`

```
fut = make_ready_future(42)      ⋮ int → future<int>
opt = make_optional(42)          ⋮ int → optional<int>
lst = list<string>{"Meaning"}    ⋮ string → list<string>
```

Monads as containers

Transform (map) method

```
(C<From>, function<To(From)>) -> C<To>
```

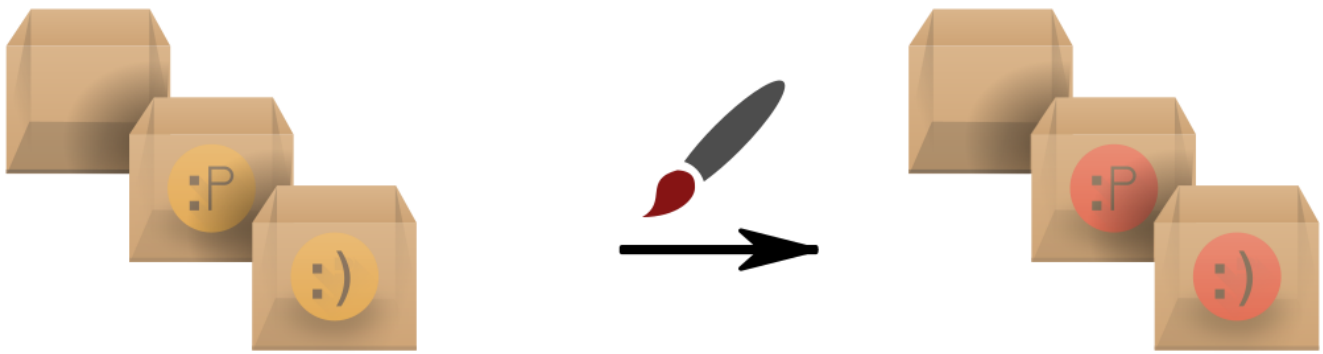



```
fut | transform(to_string)  : future<int> → future<string>
opt | transform(factoriel)  : optional<int> → optional<int>
lst | transform(to_upper)   : list<string> → list<string>
```

Monads as containers

Transform (map) method

```
(C<From>, function<To(From)>) -> C<To>
```

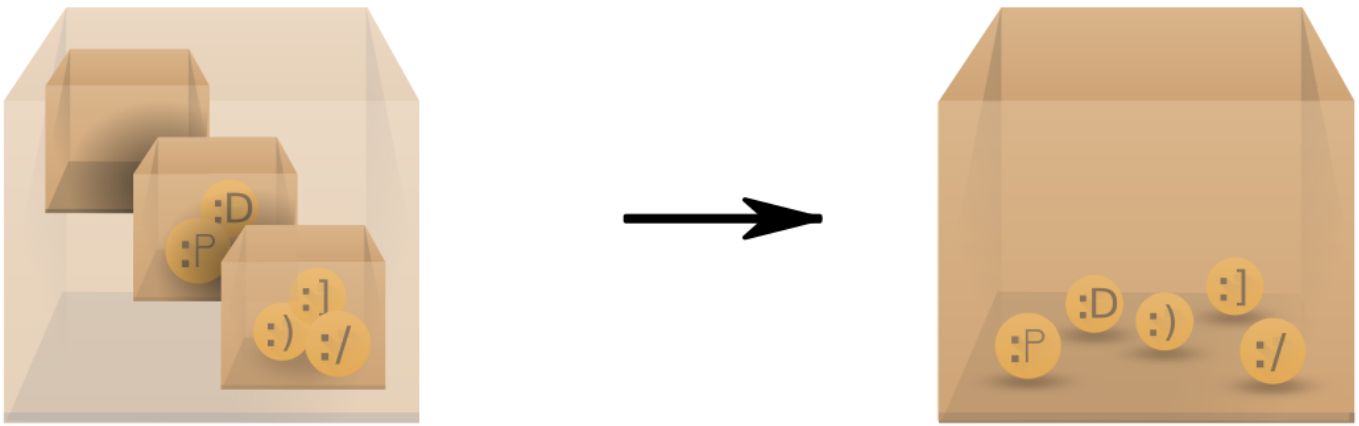



```
fut | transform(to_string)  : future<int> → future<string>
opt | transform(factoriel)  : optional<int> → optional<int>
lst | transform(to_upper)   : list<string> → list<string>
```

Monads as containers

Flatten method

```
(C<C<T>>) -> C<T>
```



- future<future<int>> → future<int>
- optional<optional<int>> → optional<int>
- list<list<string>> → list<string>

Optional, or anything else

```
// config_value<T>(string) → optional<T>

optional<int> year() {
    auto year =
        co_await config_value<int>("year");
    co_return (year < 100) ?
        year + 1900 : year;
}
```

Optional, or anything else

```
// config_value<T>(string) → optional<T>

optional<int> year() {
    auto year =
        co_await config_value<int>("year");
    co_return (year < 100) ?
        year + 1900 : year;
}

optional<int> year() {
    return config_value<int>("year") |
        transform([] (int year) {
            return (year < 100) ?
                year + 1900 : year;
        });
}
```

Optional, or anything else

```
// config_value<T>(string) → optional<T>

optional<int> year() {
    auto year = co_await config_value<int>("year");
    auto animal = co_await animal_for_year(year);
    auto picture = co_await animal_picture(animal);
    co_return picture;
}

optional<int> year() {
    return config_value<T>("year")
         | transform(animal_for_year)
         | transform(animal_picture); ???
}
```

Optional, or anything else

```
// config_value<T>(string) → optional<T>

optional<int> year() {
    auto year = co_await config_value<int>("year");
    auto animal = co_await animal_for_year(year);
    auto picture = co_await animal_picture(animal);
    co_return picture;
}

optional<int> year() {
    return config_value<int>("year")
         | mbind(animal_for_year)
         | mbind(animal_picture);
}
```

Optional, or anything else

```
// config_value<T>(string) → optional<T>

optional<int> year() {
    auto year = co_await config_value<int>("year");
    auto animal = co_await animal_for_year(year);
    auto picture = co_await animal_picture(animal);
    co_return picture;
}

optional<int> year() {
    return config_value<int>("year")
           >>= animal_for_year
           >>= animal_picture;
}
```

N4287

```
M<T> f()
{
    auto x = co_await f1();
    auto y = co_await f2();
    co_return g(x, y);
}
```

f1() → M1<A>

f2() → M2<B>

g(A,B) → T

Comparison

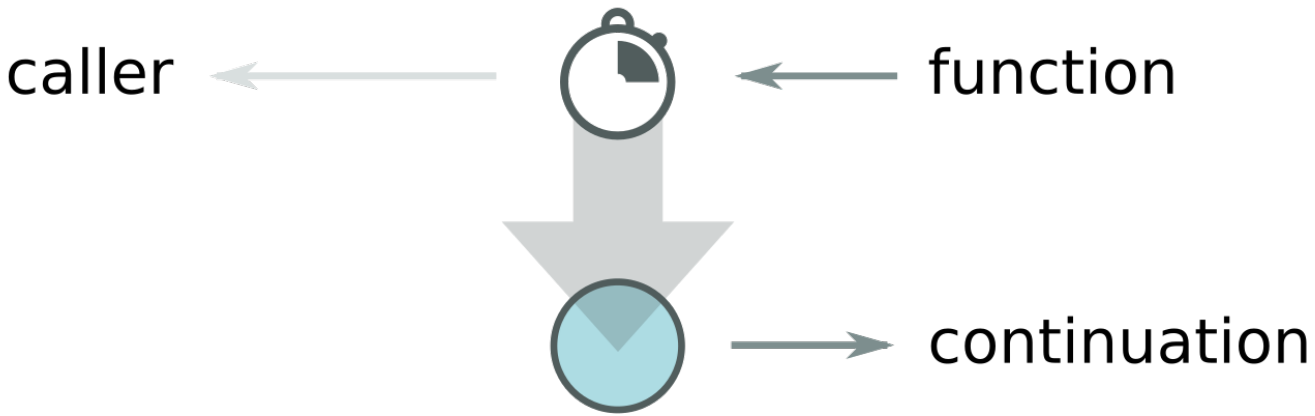
co_await

optional<T>
expected<T,E>
future<T>
(2)

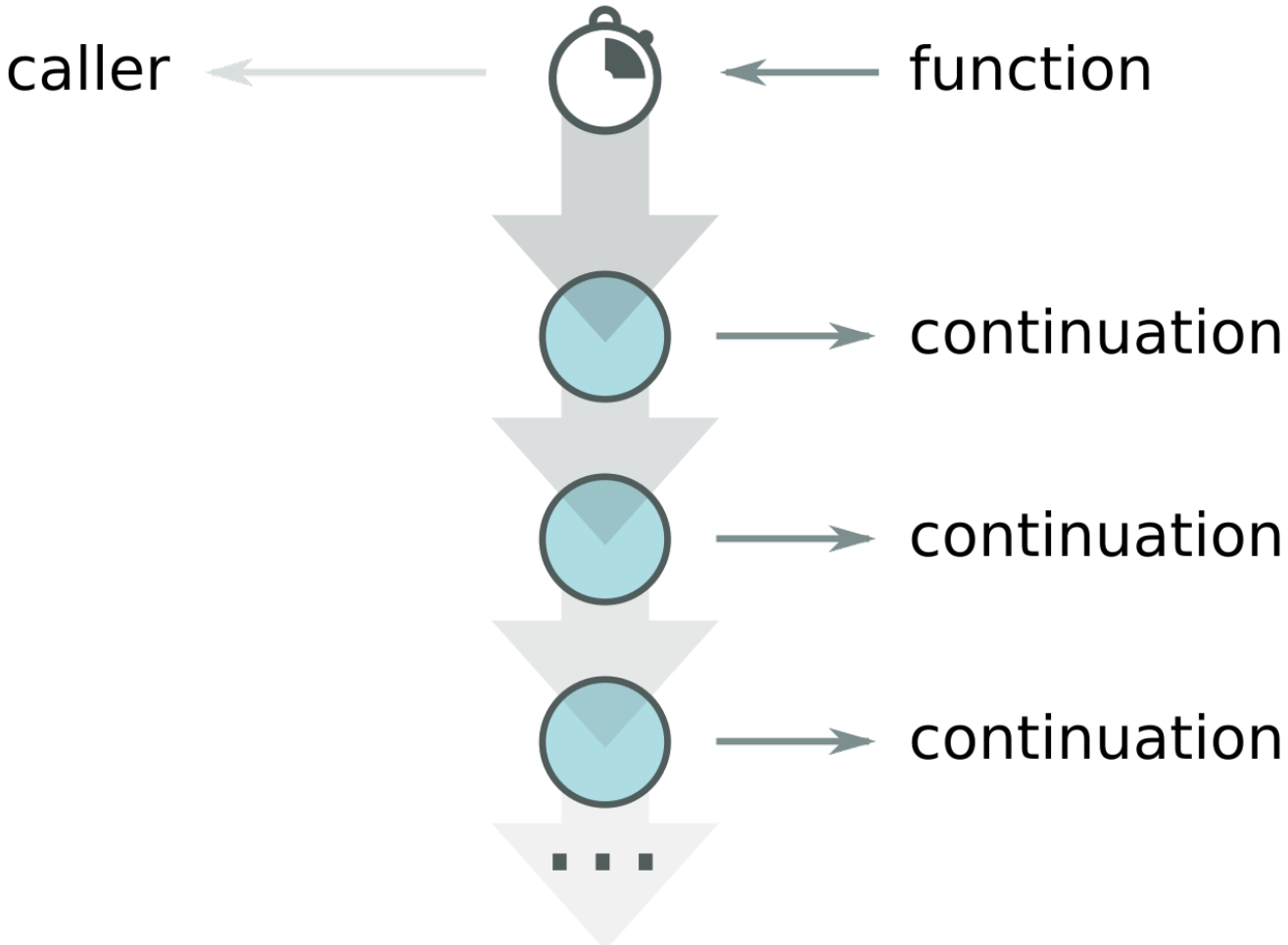
ranges

| optional<T>
| expected<T,E>
| (1)
| list<T>, vector<T>, ...

Just passing our time



Oh we'll keep on trying



Till the end of time

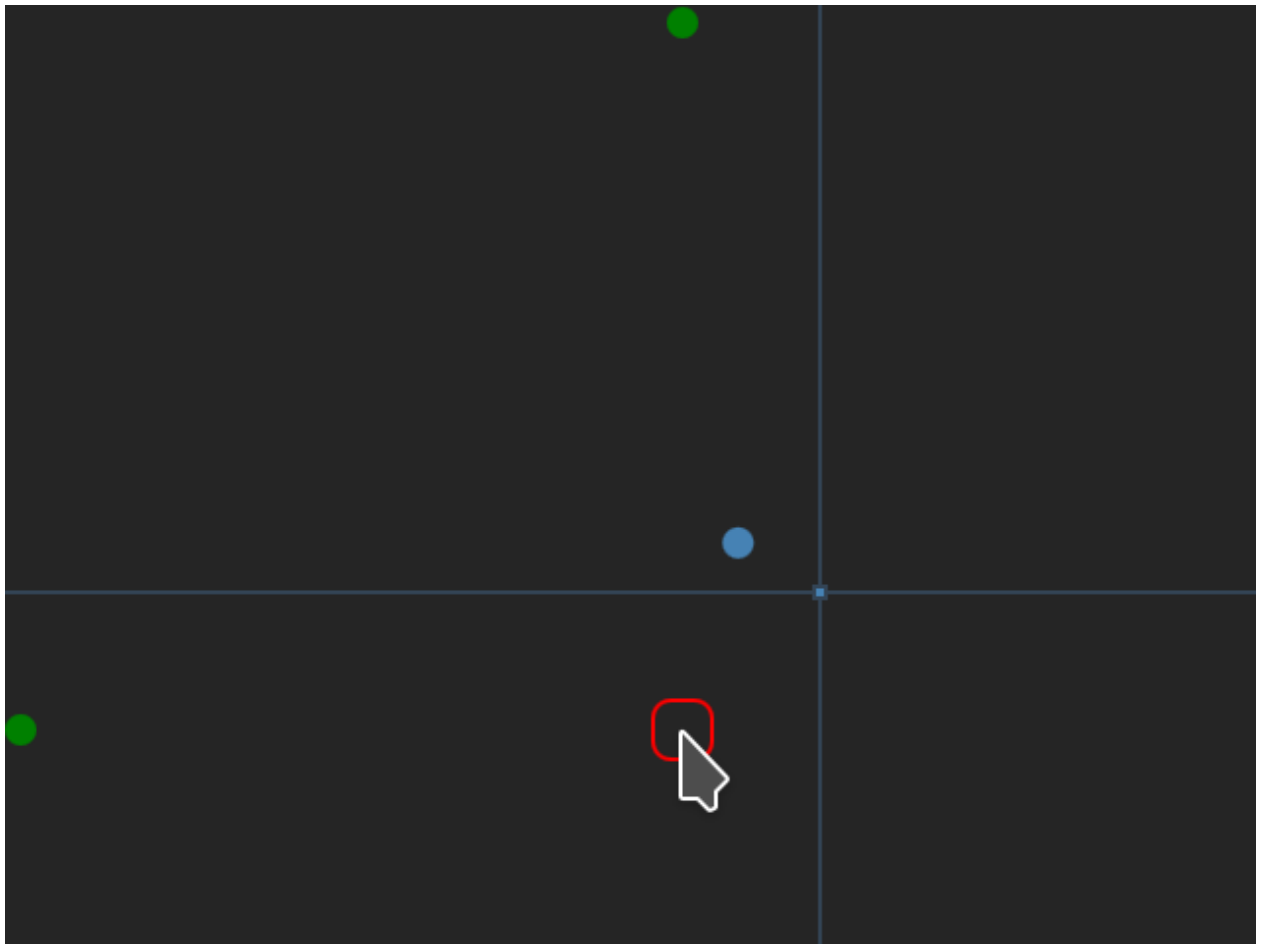
- Message passing:
  `continuation!newClientMessage`
- Call-callback:
  `onNewMessage(continuation)`
- Signals-slots:
```
connect(socket,   &Socket::newConnection,
        receiver, &Receiver::continuation)
```
- Any data collection:
  `for_each(xs, continuation)`

Stream transformation

Streams can only be transformed with algorithms that accept input ranges, since we don't have all the items. We don't even know when (if) they will end.

`map`, `bind`, `filter`, `take`, `drop`, etc.

Stream transformation



Comparison, again

- Pulling events from the stream
- Pushing events to the stream

Answers? Questions! Questions? Answers!

Kudos:

Friends at KDE, Dr Saša Malkov, Dr Zoltán Porkoláb

Further reading and watching:

- Iterators Must Go, Andrei Alexandrescu
- Value Semantics and Range Algorithms, Chandler Carruth
- Await 2.o, Gor Nishanov
- Systematic Error Handling in C++, Andrei Alexandrescu
- Learn You a Haskell for Great Good!, Miran Lipovača
  (learning Haskell is fun and can be useful)