Cpp North 2023

And Then() Some(T)

Victor Ciura

Abstract

Don't look in the box!

Forget about Monads and burritos - let's get practical and see how C++ got more functional by way of Rust Option(T) and Haskell Maybe.

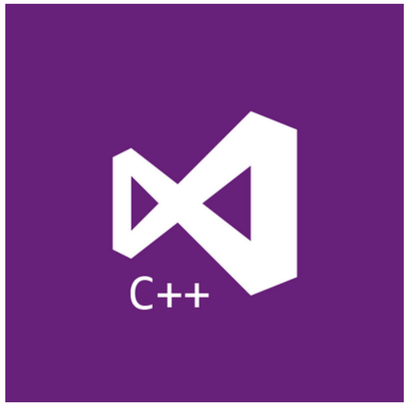
Can we write cleaner code using continuations? Let's explore patterns of using C++23 std::optional and std::expected.

See how combinators and higher-order functions can be used to manage control flow in a modular fashion, by building pipelines of computation yielding values.

About me

Advanced Installer

Clang Power Tools



Visual C++

@ciura_victor

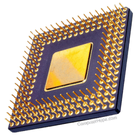
Paradox of Programming

Paradox of Programming

Is it easier to think like a machine,

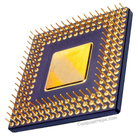
than to do some simple math?

Paradox of Programming

 **Machine** /  **Human** impedance mismatch:

- Local / Global perspective
- Progress / Goal oriented
- Detail / Idea
- Vast / Limited memory
- Pretty reliable / Error prone
- Machine language / Mathematics / Logic

Paradox of Programming

 **Machine** /  **Human** impedance mismatch:

- Local / Global perspective
- Progress / Goal oriented
- Detail / Idea
- Vast / Limited memory
- Pretty reliable / Error prone
- Machine language / Mathematics / Logic

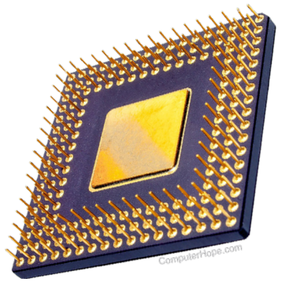
Bartosz Milewski

youtube.com/watch?v=JH_Ou17_zyU

Imperative

HOW

The computation method is
variable assignment



Declarative

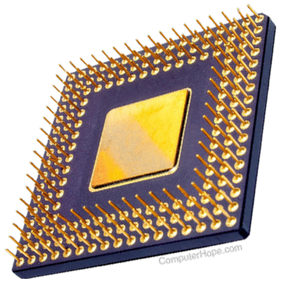
WHAT

The computation method is
logic & rules

Imperative

HOW

The computation method is
variable assignment



Functional

WHAT

The computation method is
function application

Goal:
Minimizing moving parts

Michael Feathers
@mfeathers

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

3:27 PM - 3 Nov 2010

What is Functional Programming ?

- Functional programming is a **style** of programming in which the basic method of computation is the ***application of functions*** to arguments

- A functional **language** is one that supports and encourages the ***functional style***

Are you already using std::optional ?

Would you want to write exception-less error handling with C++23 std::expected ?

You only need to understand 2 functions to write C++ in a functional style

`transform()`

`and_then()`

And Then() Some(T)

July 2023

Victor Ciura
Principal Engineer
Visual C++

@ciura_victor
@ciura_victor@hachyderm.io

A functional language is one that supports and encourages the `functional style`

What do you mean ?

pipelines

ranges

optional

IO monad

algorithms

Maybe | Just

lifting

monoids

lambdas & closures

fold

values types

lazy evaluation

declarative vs imperative

algebraic data types

monads

higher order functions

map

composition

pattern matching

FP

expressions vs statements

pure functions

currying

partial application

category theory

recursion

Not a talk about...

Haskell

But...

Most of the "new" ideas and innovations in modern programming languages are actually very old...

1930s Alonzo Church develops the lambda calculus, a simple but powerful *theory of functions*

1950s John McCarthy develops Lisp, the *first functional language*, with some influences from the lambda calculus, but retaining *variable assignments*

1960s Peter Landin develops ISWIM, the first *pure functional language*, based strongly on the lambda calculus, with *no assignments*

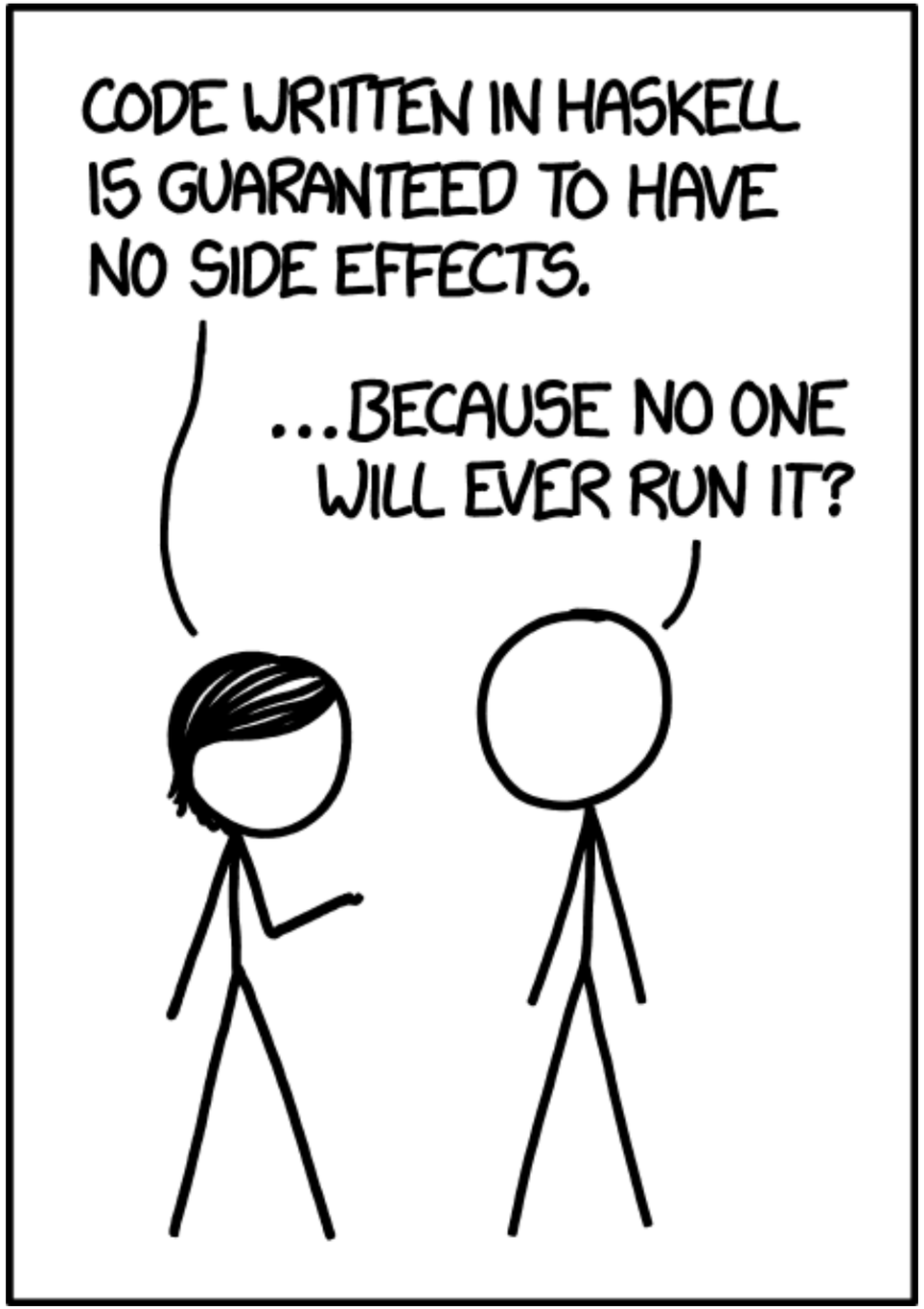
1970s John Backus develops FP, a functional language that emphasizes *higher-order functions* and reasoning about programs

1970s Robin Milner and others develop ML, the first modern functional language, which introduced *type inference* and *polymorphic types*

70-80s David Turner develops a number of lazy *functional languages*, culminating in the Miranda system

1987 An international committee starts the development of Haskell, a standard *lazy functional language*

1990s Phil Wadler and others develop type classes and monads, two of the main innovations of Haskell



xkcd.com/1312/

Why (not) Haskell ?

If Haskell is so great, why hasn't it taken over the world?

Why (not) Haskell ?

If Haskell is so great, why hasn't it taken over the world?

My claim is that it has.

But not as a Roman legion loudly marching in a new territory, rather as distributed Trojan horses popping in at the gates, masquerading as modern features or novel ideas in today's mainstream languages.

Modern C++ is functional

Functional Programming ideas that have been around for over 40 years are rediscovered to solve our current software complexity problems.

Indeed, contemporary C++ has become more functional.

From mundane concepts like lambdas & closures, std::function, values, ADT, to composability of STL algorithms, lazy ranges, folding, mapping, partial application, higher-order functions or even monads such as optional, future, expected ...

A Taste of Haskell

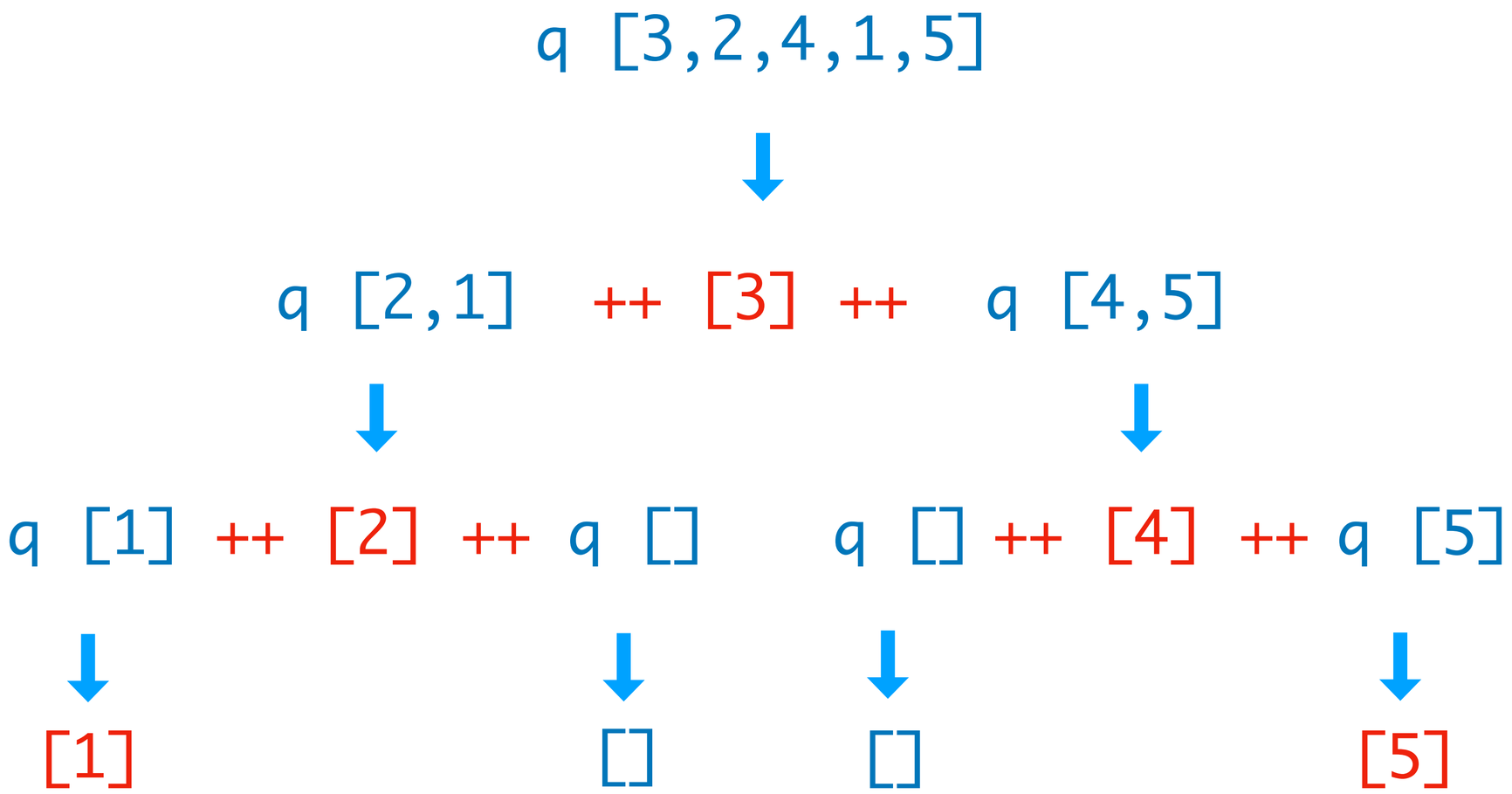
```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
               ys = [a | a ← xs, a ≤ x]
               zs = [b | b ← xs, b > x]
```

What does `f` do ?

Quick Sort

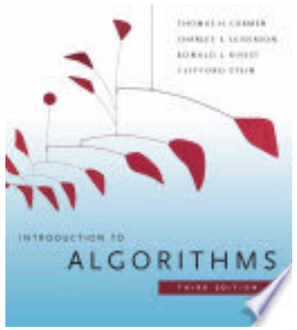
```
qsort :: Ord a ⇒ [a] → [a]
qsort []     = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

Quick Sort



Quick Sort

```
void quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```



pseudo-code

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

True Story

1986:

Donald Knuth was asked to implement a program for the *"Programming pearls"* column in the **Communications of ACM** journal.

The task:

Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

True Story

1986:

Donald Knuth was asked to implement a program for the *"Programming pearls"* column in the **Communications of ACM** journal.

The task:

Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.

His solution written in `Pascal` was **10 pages** long.

True Story

Doug McIlroy

wikipedia.org/wiki/Douglas_McIlroy

True Story

Doug McIlroy

His response was a 6-line shell script that did the same:

```
tr -cs A-Za-z '\n' |
    tr A-Z a-z     |
    sort           |
    uniq -c        |
    sort -rn       |
    sed ${1}q
```

wikipedia.org/wiki/Douglas_McIlroy

It's all about | pipelines

Taking inspiration from Doug McIlroy's UNIX shell script,

write an algorithm in `your favorite programming language`,

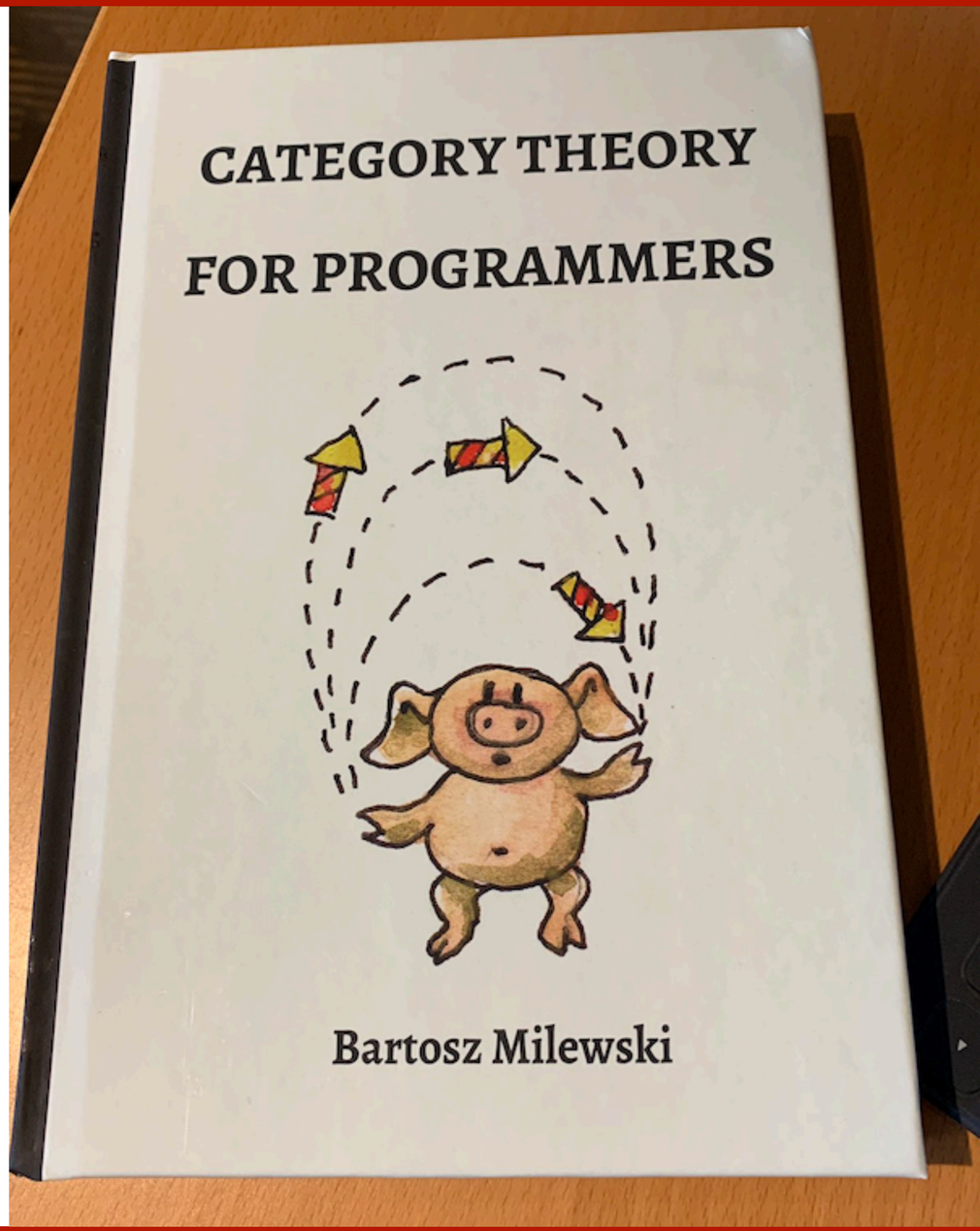
that solves the same problem: word frequencies

I'm hooked.

Where do I start?

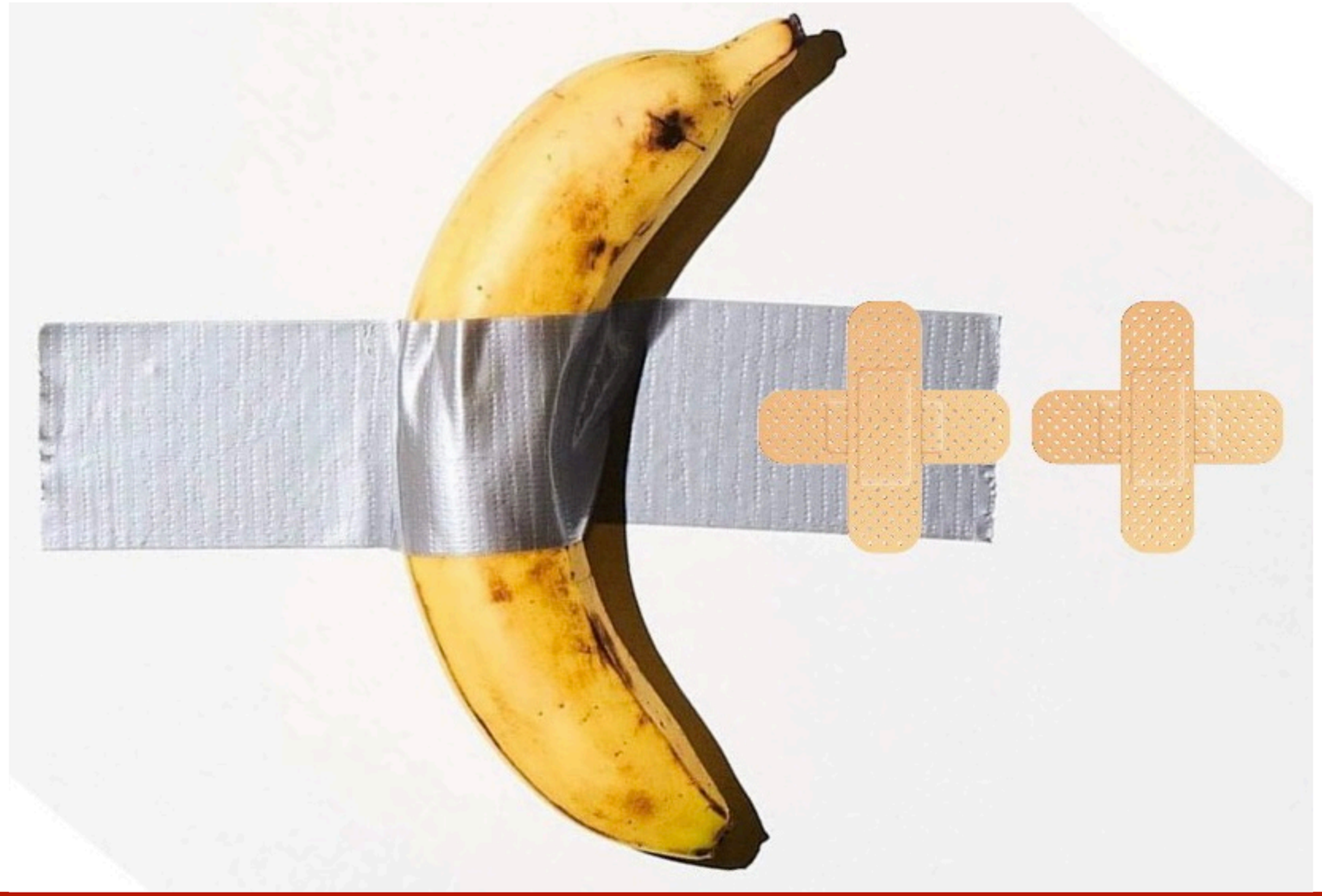
The Book

Bartosz Milewski
@BartoszMilewski



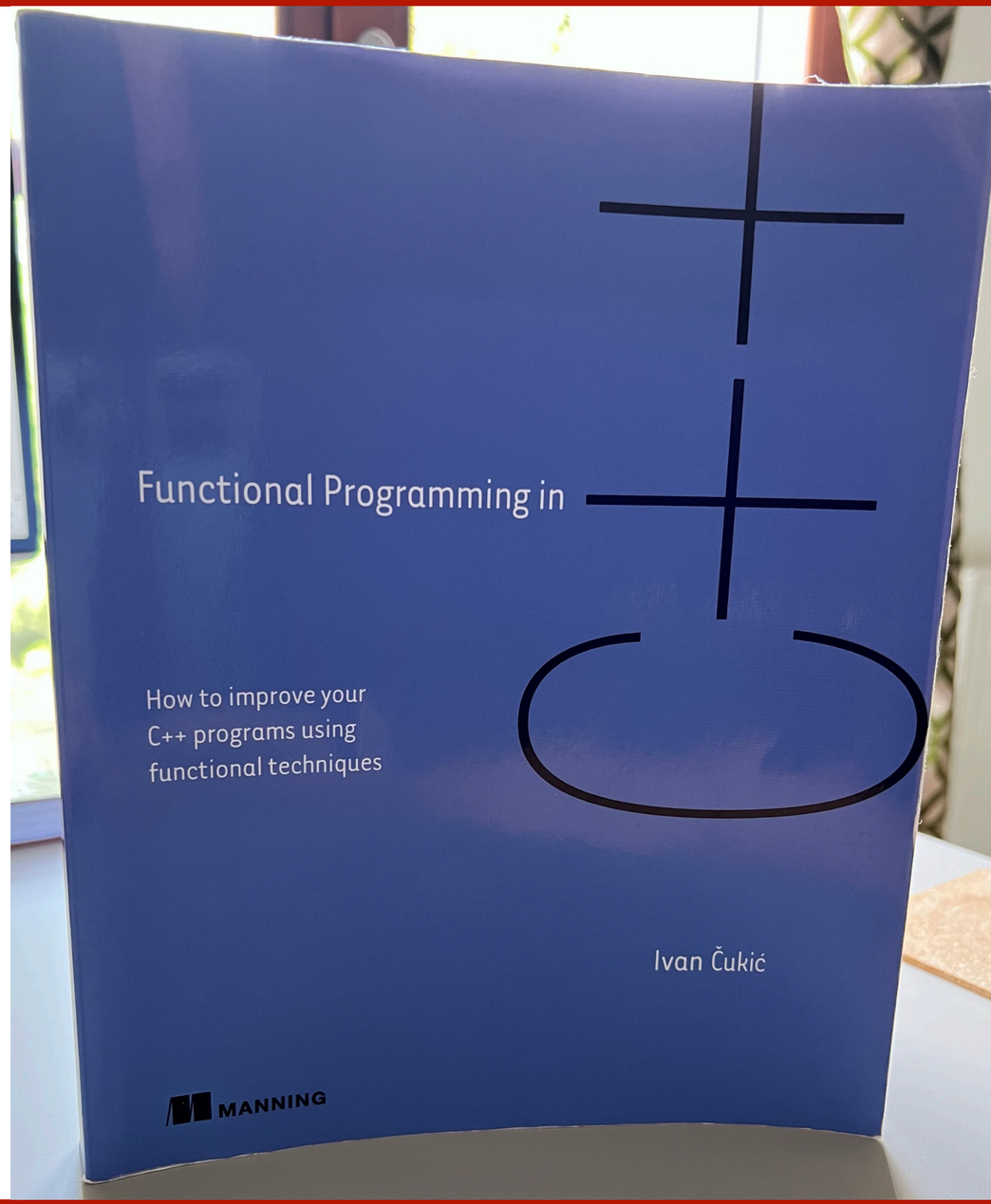
github.com/hmemcpy/milewski-ctfp-pdf

Functional Programming but... in C++ 😳



The Book

Ivan Čukić
@ivan_cukic



amazon.com/Functional-Programming-programs-functional-techniques

You only need to understand 2 functions to write C++ in a functional style

`transform()`

`and_then()`

Need a lift?

Lift ⬆️

Higher-Order Functions

boost::hof

boost.org/doc/libs/develop/libs/hof/doc/html/doc/

Need a lift?

A C++17 library of simple constexpr higher order functions of predicates and for making functional composition easier.

These help reduce code duplication and improve clarity, for example in code using STL <algorithm>

github.com/rollbear/lift

Need a lift?

Higher order functions

- `equal`
- `not_equal`
- `less_than`
- `less_equal`
- `greater_than`
- `greater_equal`
- `negate`
- `compose`
- `when_all`
- `when_any`
- `when_none`
- `if_then`
- `if_then_else`
- `do_all`

Need a lift?

```
struct Employee {
  std::string name;
  unsigned    number;
};

const std::string& select_name(const Employee& e) { return e.name; }
unsigned select_number(const Employee& e) { return e.number; }

std::vector<Employee> staff;

// sort employees by name
std::sort(staff.begin(), staff.end(),
          lift::compose(std::less<>{}, select_name));

// retire employee number 5
auto i = std::find_if(staff.begin(), staff.end(),
                      lift::compose(lift::equal(5), select_number));
if (i != staff.end()) staff.erase(i);
```

Need a lift?

If you're using C++20 ranges you can get this (and more).

Projections... Oh my!

Need a lift?

Lifts overloaded functions named 'X' to one callable that can be used with other higher order functions.

```
#define LIFT_THRICE(...)                    \
       noexcept(noexcept(__VA_ARGS__))      \
        -> decltype(__VA_ARGS__)            \
        {                                   \
          return __VA_ARGS__;               \
        }

#define LIFT_FWD(x) std::forward<decltype(x)>(x)

#define LIFT(lift_func) [](auto&& ... p)
                        LIFT_THRICE(lift_func(LIFT_FWD(p)...))
```

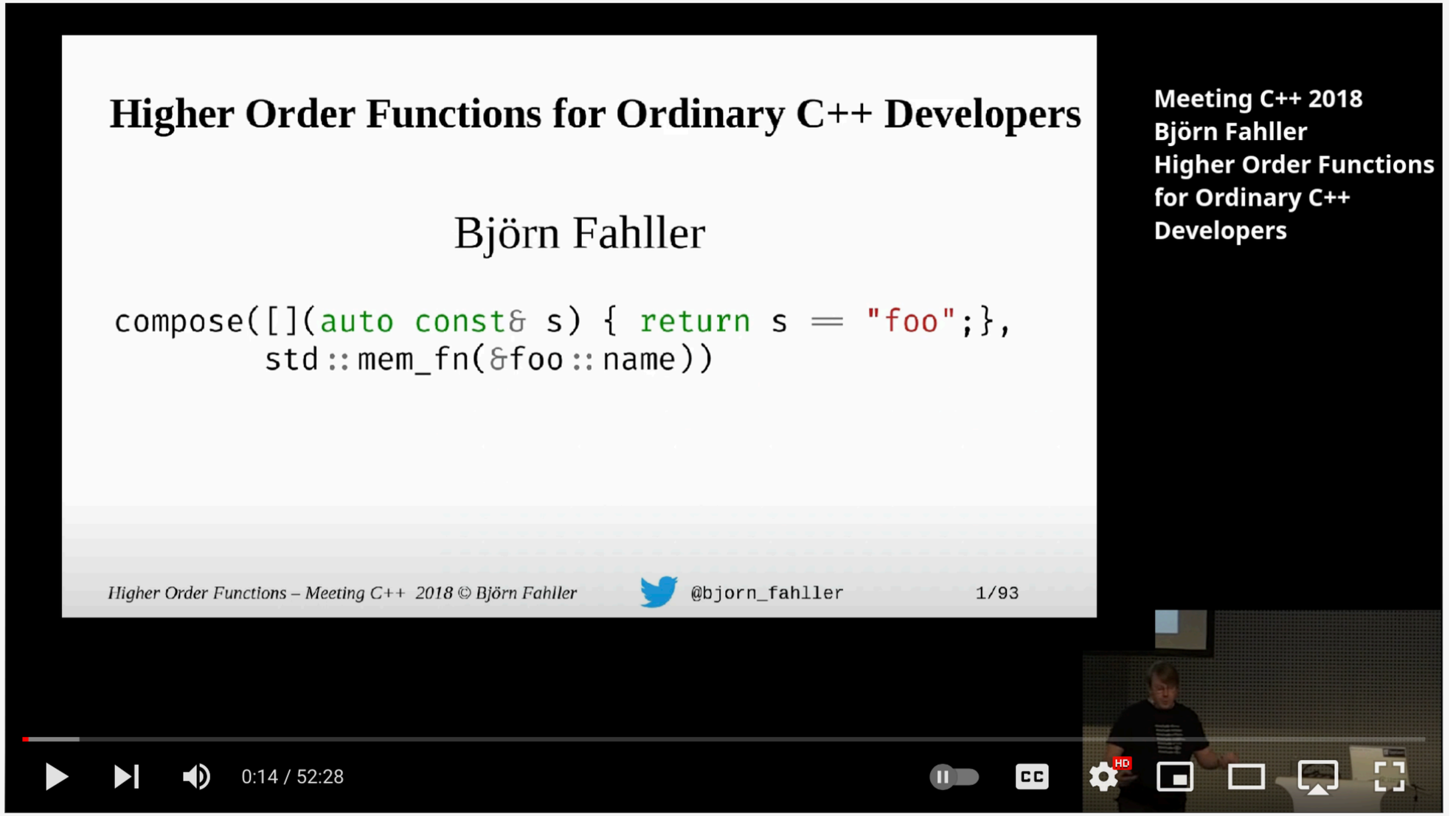
Need a lift?

Lifts overloaded functions named 'X' to one callable that can be used with other higher order functions.

```
std::vector<int> vi;
... ⬇️
std::vector<std::string> vs;

std::transform(std::begin(vi), std::end(vi),
               std::back_inserter(vs),
               LIFT(std::to_string)); //lift overloaded set of 9 functions
```

Lifting overload sets into objects: **wg21.link/p0834**

Need a lift?



Higher Order Functions for ordinary developers - Björn Fahller - Meeting C++ 2018

youtube.com/watch?v=qL6zUn7iiLg

Boxes 

Type Constructors

There are various ways to hide 📦 a value:

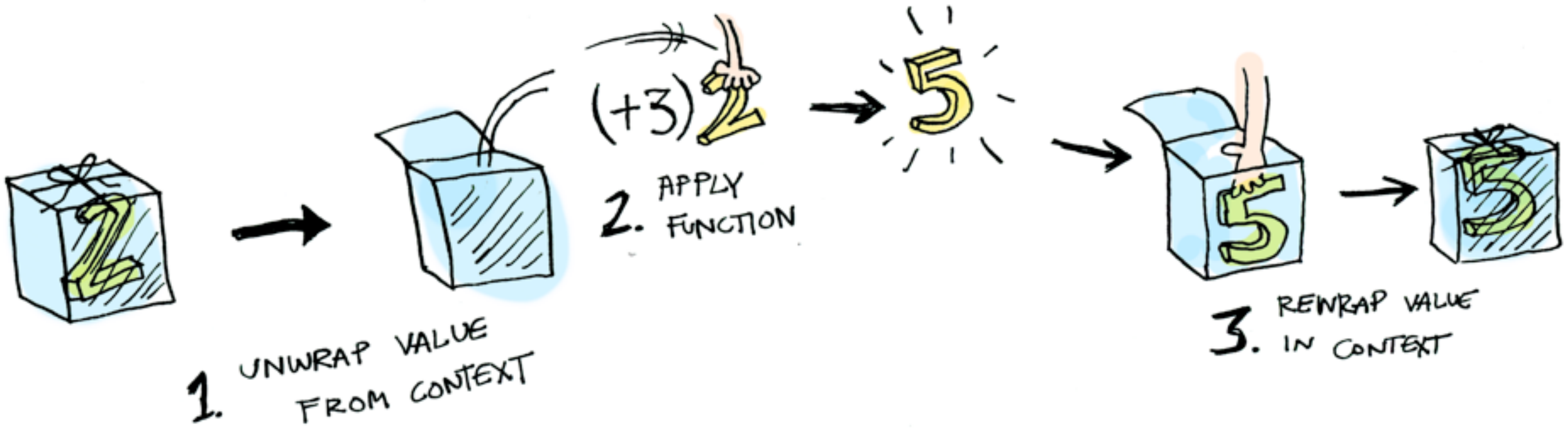
- `unique_ptr<T> p;`
- `shared_ptr<T> p;`
- `vector<T> v;`
- `optional<T> o;`
- `function<T(int)> f;`

Access the value within:

- `*p| p.get()`
- `*p| p.get()`
- `v[0] | *v.begin()`
- `*o| o.value()`
- `f(5)`

Functor | Applicative | Monad

Performing actions on the hidden value, without breaking the 📦 BOX.



Aditya Bhargava

adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures

📦 The Box

`std::optional` can simplify code

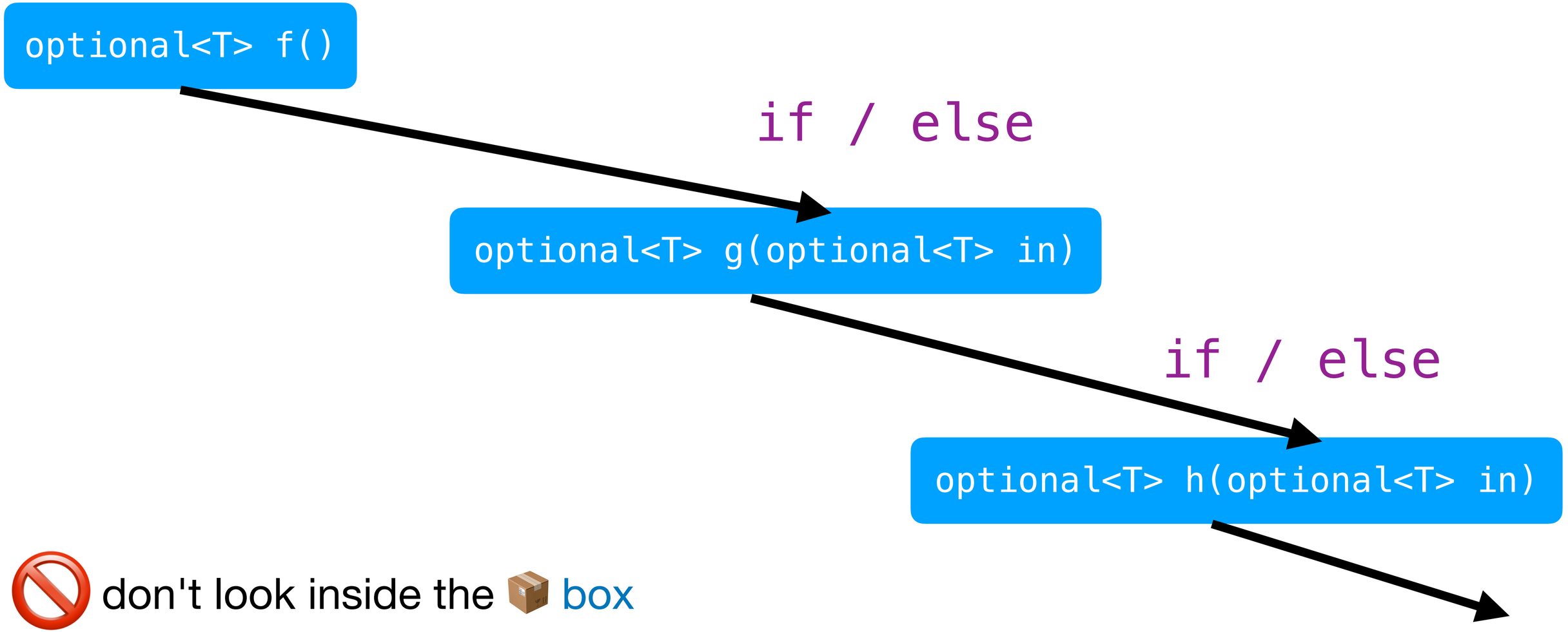
- don't look inside the 📦 box (unwrap)
- don't use optional for error handling
- when in doubt, draw inspiration from other languages:

  Haskell (`Maybe`) or Rust (`Option<T>`)

📦 The Box

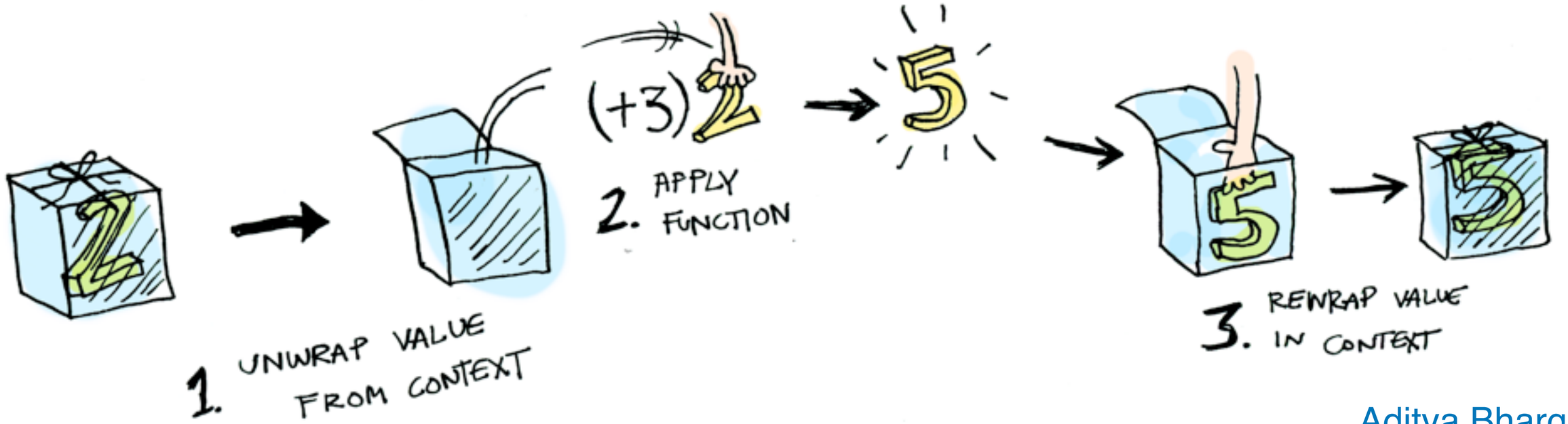
doc.rust-lang.org/rust-by-example/error/option_unwrap

📦 The Box



🚫 don't look inside the 📦 box

The Box



Aditya Bhargava

adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures

Example

Calling the a function on the `std::string` value inside the `std::optional` box.

```
string capitalize(string str);
...

optional<string> str = ...; // from an operation that could fail

string cap;
if (str)
  cap = capitalize(str.value()); // capitalize(*str);
```

Example

Calling the a function on the `std::string` value inside the `std::optional` box.

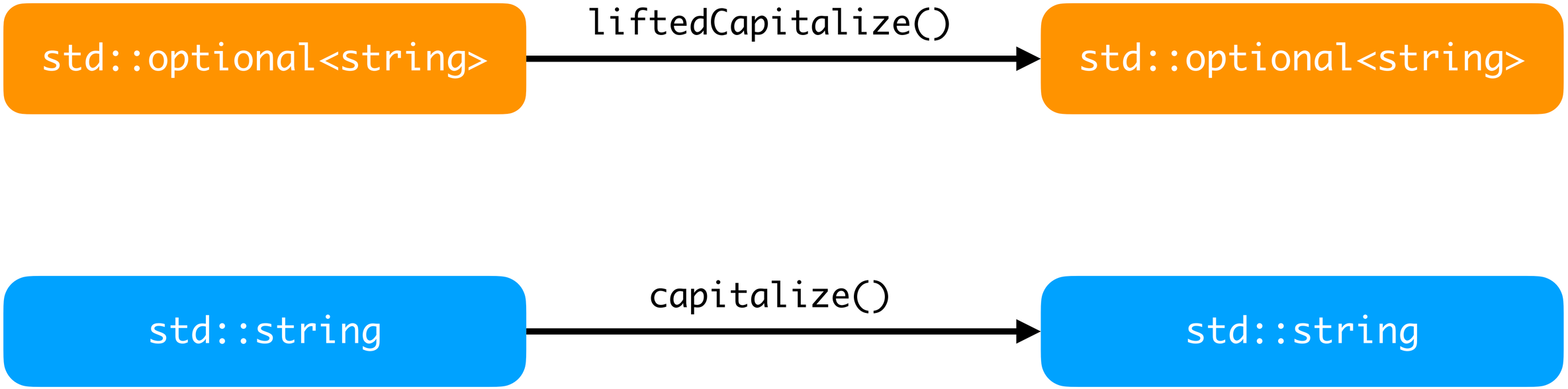
```
string capitalize(string str);
...

optional<string> str = ...; // from an operation that could fail

optional<string> cap;
if (str)
  cap = capitalize(str.value()); // capitalize(*str);
```

Lifting `capitalize()`

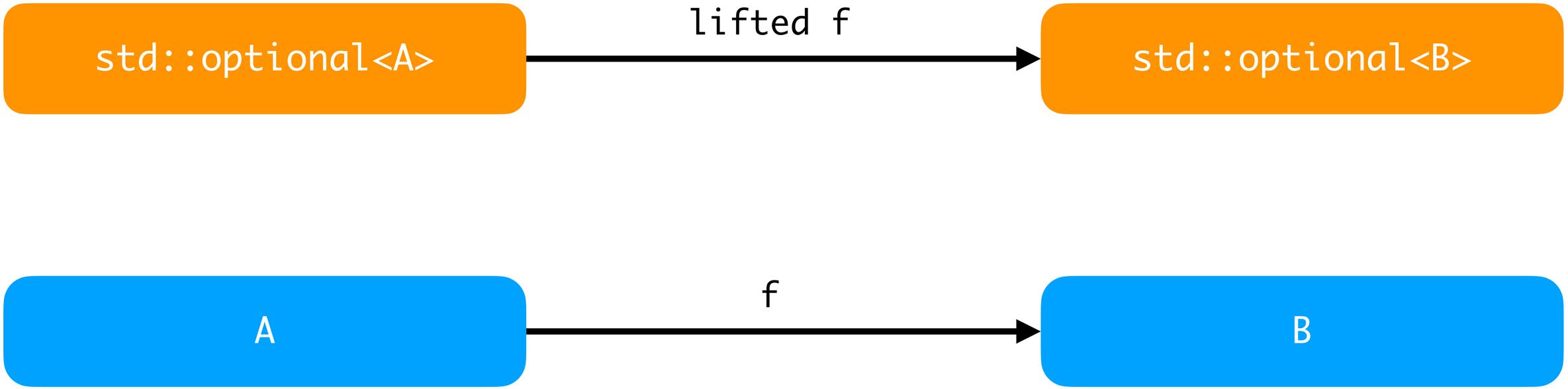
Lifted `capitalize()` operates on `optional<string>` and produces `optional<string>`

```
optional<string> liftedCapitalize(const optional<string> & s)
{
  optional<string> result;
  if (s)
    result = capitalize(*s);

  return result;
}
```

Lifting capitalize()



Lifting any function



Lifting any function

"Lifted `f`" operates on `optional<A>` and produces `optional<B>`

```
template<class A, class B>
optional<B> fmap(function<B(A)> f, const optional<A> & o)
{
  optional<B> result;
  if (o)
    result = f(*o); // wrap a <B>

  return result;
}
```

Lifting any function (take 2)

```
template<typename T, typename F>
auto fmap(const optional<T> & o, F f) -> decltype( f(o.value()) )
{
  if (o)
    return f(o.value());
  else
    return {}; // std::nullopt
}
```

Lifting a function to a vector

"Lifted `f`" operates on `vector<A>` and produces `vector<B>`

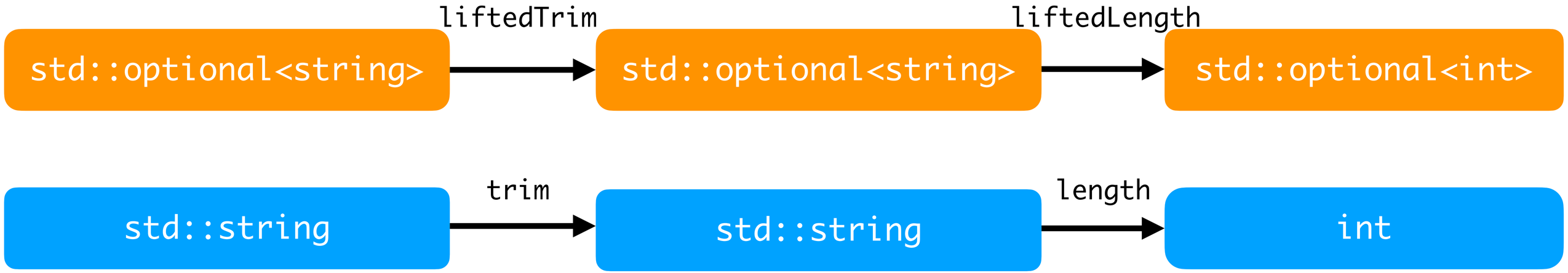
```
template<class A, class B>
vector<B> fmap(function<B(A)> f, vector<A> v)
{
  vector<B> result;
  result.reserve(v.size());
  std::transform(v.begin(), v.end(), back_inserter(result), f);
  return result;
}
```

```
vector<string> names{ ... };

vector<int> lengths = fmap<string, int>(&length, names);
```

Composition of lifted functions

The real power of lifted functions shines when composing functions.

```
optional<string> str{"  Some text  "};

auto len = fmap<string, int>(&length,
                             fmap<string, string>(&trim, str));
```



Composition Example

Let’s build a symbol table for a debugged program.

```
optional<int64_t> current_pc = ... ; // function address
...

optional<string> debug_location()
{
  if (!current_pc)
    return {};

  const auto function = dsym::load_symbol(current_pc.value());
  if (!function)
    return {};

  return dsym::to_string(function.value()); // function name
}
```

Composition Example (take 2)

Let’s build a symbol table for a debugged program.

```
optional<int64_t> current_pc = ... ; // function address
...

optional<string> debug_location()
{
  return fmap(
    fmap(current_pc, dsym::load_symbol),
    dsym::to_string
  );
}
```

Composition Example (take 3)

We could create an `fmap` transformation that has the pipe | syntax, like ranges:

```
optional<int64_t> current_pc = ... ; // function address
...

optional<string> debug_location()
{
  return current_pc
           | fmap(dsym::load_symbol)
           | fmap(dsym::to_string);
}
```

Functor (recap)

Type constructor

- create a **`box`** type that wraps another type
- encapsulates the values of another type into a *context*

Function lifting

- create a *higher-order* function (eg. `fmap`)
- for any function `A->B` create a function `box<A> -> box<B>`

Why?

- no need to break encapsulation (no peek in 📦)
- better composition (chaining, continuation)

📦 The Box

Monadic `std::optional` (C++23 P0798)

```
optional<int> string_view_to_int(string_view sv)
{
  const auto first = sv.data();
  const auto last  = first + sv.size();
  int val = -1;
  const auto result = std::from_chars(first, last, val);

  if (result.ec == errc{} && result.ptr == last)
    return val;
  else
    return nullopt;
}
```

📦 The Box

Monadic `std::optional` (C++23 P0798)

```
cout << string_view_to_int(sv)
          .and_then([=](int val) -> optional<int> {
                    const int logs = clamp(val, 0, max_logs);
                    if (logs > 0)
                      return logs;
                    else
                      return std::nullopt;
                  })
          .transform([](int val) {
                     return std::format("Collecting in {} logs.", val);
                   })
          .or_else([] {
                    return optional<string>{"Log error"};
                  })
          .value()
```

Heritage

```
enum Option<T> {
    None,
    Some(T),
}
```

```
let second = ["Haskell", "Rust"].get(1);
println!("{:?}", second); // prints: Some("Rust")
```

```
let langs = ["C++", "Rust", "Carbon", "Val"];
let successor_lang : Option<&i32> = langs.get(4);
println!("{:?}", successor_lang); // prints: None
```

Heritage

```haskell
data Maybe a = Just a | Nothing


getFirst :: [a] -> Maybe a
getFirst (x : _) = Just x
getFirst [] = Nothing


print $ getFirst ["Haskell", "Rust", "C++"]
-- prints: Just "Haskell"

print $ getFirst []
-- prints: Nothing
```

Heritage / Concepts

`transform()`

`and_then()`

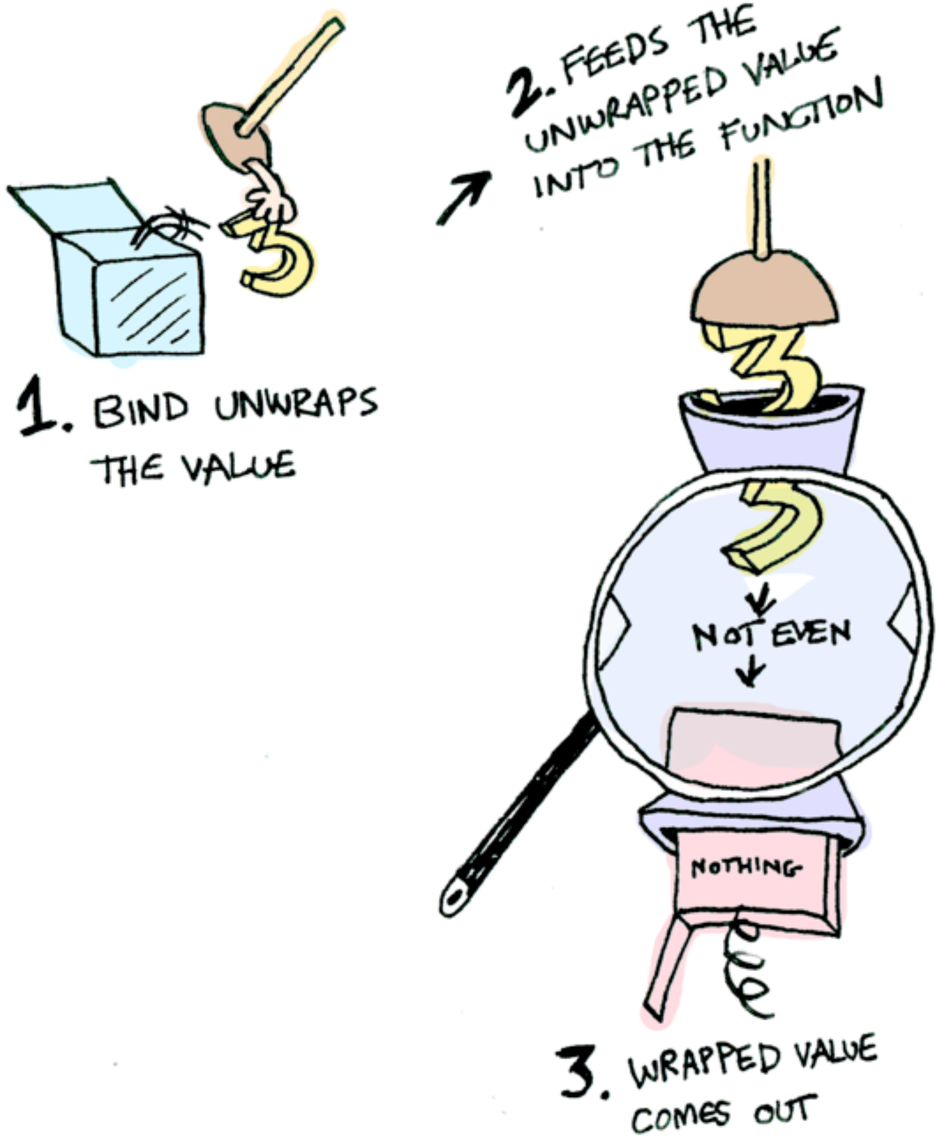
functor

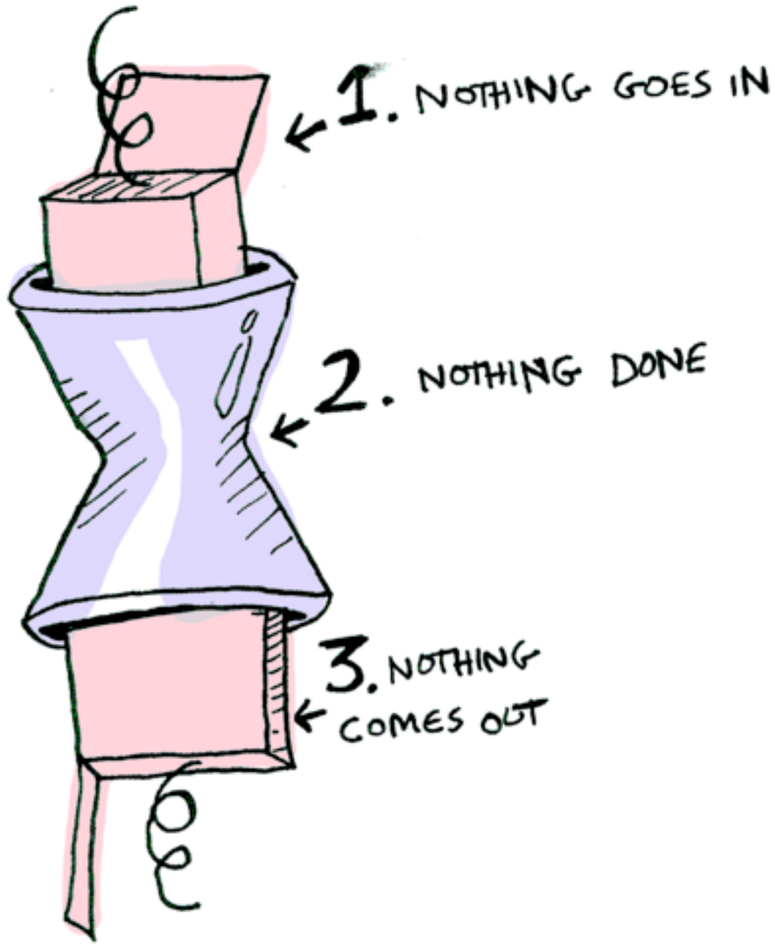
monad

`fmap`

`>>= (bind)`

Bind monad

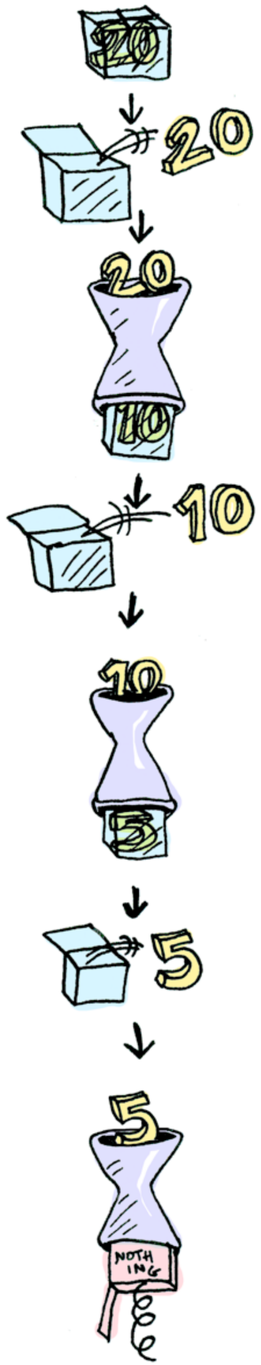




Aditya Bhargava

adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures

Bind >>= chains




```
20 >>= half >>= half >>= half
```

Aditya Bhargava

adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures

Levels...

```
auto p = create_widget(widget_type::plain)
   .or_else([]() -> std::optional<widget> {
      std::cout << "Failed to create widget";
      return std::nullopt; })
   .and_then(add_styles)
   .or_else([]() -> std::optional<widget> {
      std::cout << "Failed to add styles\n";
      return std::nullopt; })
   .and_then(add_frame)
   .or_else([]() -> std::optional<widget> {
      std::cout << "Failed to add frame\n";
      return std::nullopt; })
   .and_then(add_region)
   .or_else([]() -> std::optional<widget> {
      std::cout << "Failed to add region\n";
      return std::nullopt; })
   .transform(get_render);
```

Levels...

```
optional<widget> add_frame(optional<widget>)
{
  if(!w) return std::nullopt;
  ...
}

optional<render_surface> get_render(optional<widget> w)
{
  if(!w) return std::nullopt;
  ...
}

optional<optional<render_surface>> p = create_widget(widget_type::plain)
   .transform(get_render);
```

⚠️ If the callable passed to transform() returns a optional<T>
=> the result will be a optional<optional<T>>

Nested optionals

```
optional<optional<optional<T>>> result;
...

result.value().value().value()
```

Dude...

Dude Where's My Car - "And Then"

youtube.com/watch?v=CkdyU_eUm1U

Flattening

We can combine the two different empty states of the std::optional into just one.

=> we flatten the optional<optional<T>> into a optional<T>

⚠️ Flattening a optional<T> loses information:

- we’re squashing two distinct empty states into one.
- but without additional contexts the two empty states are the same anyway (for the caller)
- the empty state of std::optional<T> does not have any inherent meaning
  - only the *origin* gives it meaning, such as "*failed to create*" or "*failed to add frame*"
- probably OK in most cases (when context is not relevant to the caller)
- if context/reason for not having a value matters => need to propagate it

No value - Why?

std::optional - great for expressing that some operation produced no value,

but it gives us no information to help us understand why the operation failed.

No value - Why?

```
std::expected<T,E>
```

either the expected **T** value

or some **E** telling you what went wrong (why there is no value)

Expected

```
cout << string_view_to_int(sv)
          .and_then([=](int val) -> std::expected<int, ParseErr> {
                    const int logs = clamp(val, 0, max_logs);
                    if (logs > 0)
                      return logs;
                    else
                      return std::unexpected(ParseErr("out of range"));
                  })
          .transform([](int val) {
                     return val + 1; // guard against off-by-one errors 😄
                   })
          .or_else([] {
                    return std::unexpected(ParseErr("not an integer"));
                  })
          .value()
```

Heritage

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn safe_div(a: i32, b: i32) -> Result<i32, DivisionByZero> {
    match b {
        0 => Err(DivisionByZero),
        _ => Ok(a / b),
    }
}

println!("{:?}", safe_div(42, 2)); // prints: Ok(21)

println!("{:?}", safe_div(42, 0)); // prints: Err(DivisionByZero)
```

```rust
#[derive(Debug)]
struct DivisionByZero;
```

Result?

Heritage

```haskell
data Either a b = Left a | Right b
```

```haskell
safeDiv :: Int -> Int -> Either DivisionByZero Int
safeDiv x y = case y of
  0 -> Left DivisionByZero
  _ -> Right $ x `div` y

print $ safeDiv 42 2
-- prints: Right 21

print $ safeDiv 42 0
-- prints: Left DivisionByZero
```

```haskell
data DivisionByZero = DivisionByZero
  deriving (Show)
```

Availability

std::optional

- libstdc++ GCC 7
- libc++ Clang 4
- Microsoft STL VS2017 15.2

C++ 17

Availability

std::expected

- libstdc++ GCC 12
- libc++ Clang 16
- Microsoft STL VS2022 17.3

C++ 23

Availability

.then()

Monadic operations for std::optional (P0798)

- libstdc++ GCC 12
- libc++ Clang 14
- Microsoft STL VS2022 17.6

C++ 23

Monadic operations for std::expected (P2505)

- libstdc++ GCC 13
- libc++ Clang 17
- Microsoft STL VS2022 17.6

Availability

Are we there yet?

- tl::optional
  - https://github.com/TartanLlama/optional
- tl::expected
  - https://github.com/TartanLlama/expected

C++11/14/17 functional interfaces, as single-header libraries

Read more

Sy Brand

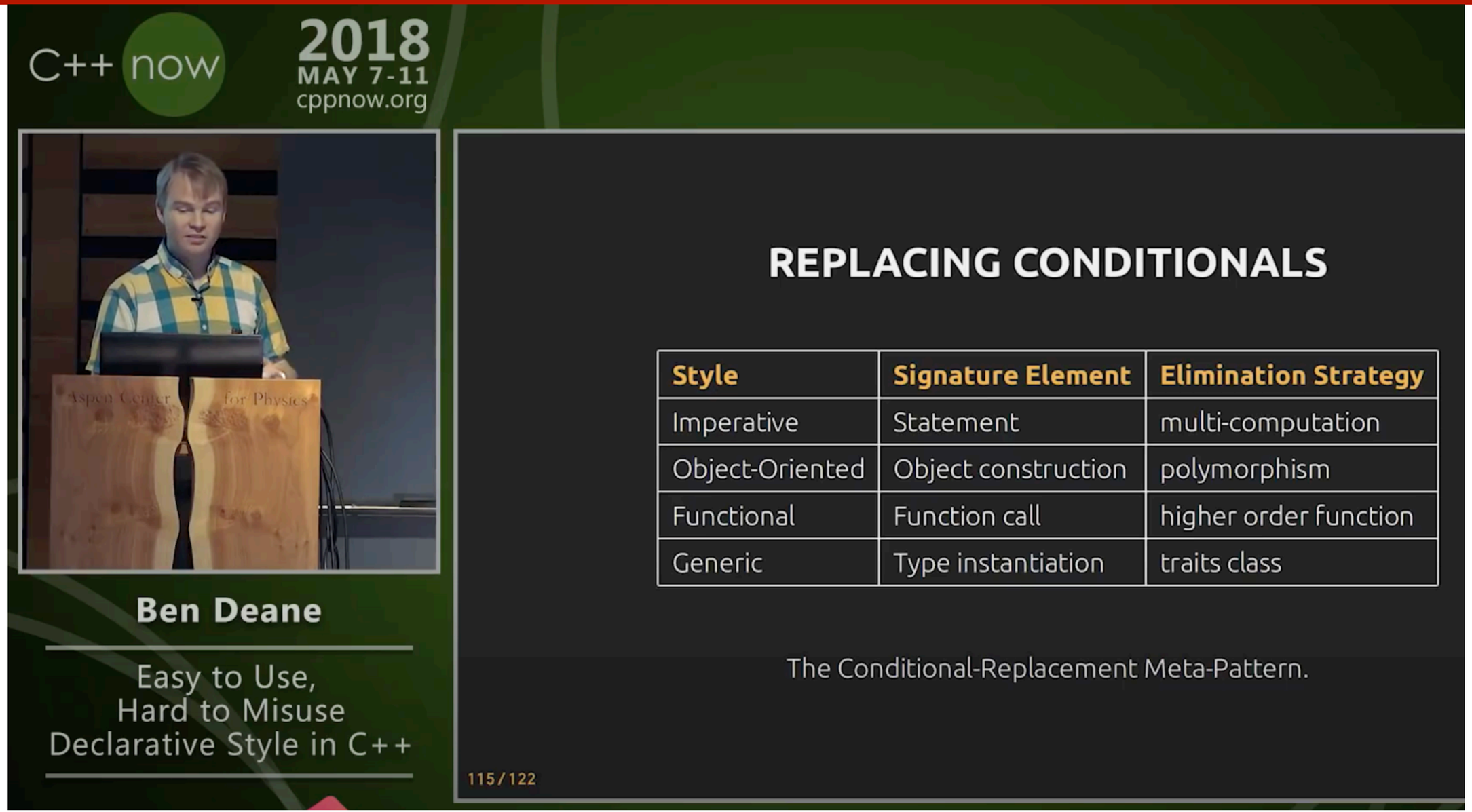
Functional exception-less error handling with C++23's optional and expected

https://devblogs.microsoft.com/cppblog/cpp23s-optional-and-expected/

Values

Expressions yield **values**, Statements do not;

Declarative style



youtube.com/watch?v=2ouxETt75R4

Must watch

youtube.com/watch?v=QthAU-t3PQ4

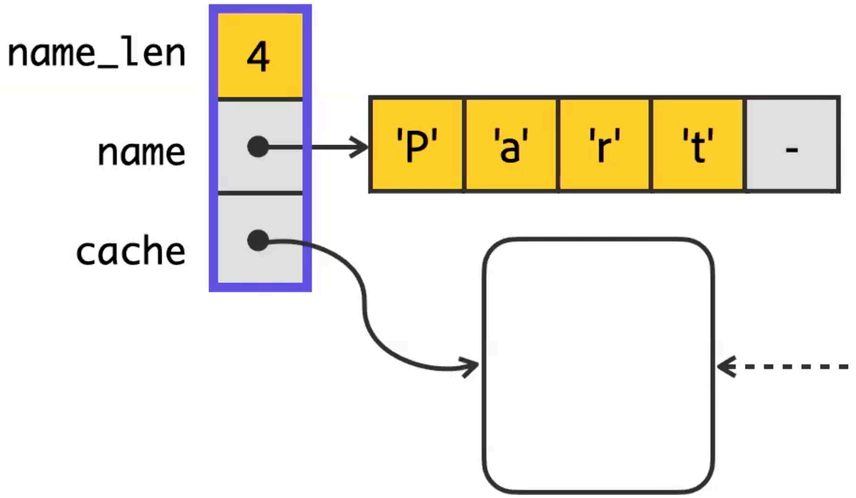
Value Semantics: Safety, Independence, Projection, & Future of Programming - Dave Abrahams CppCon 22

Values: whole-part semantics

Achieving value semantics today | decoupling an object graph

What's a value? You decide 👊

That choice determines the *meaning* of a type.



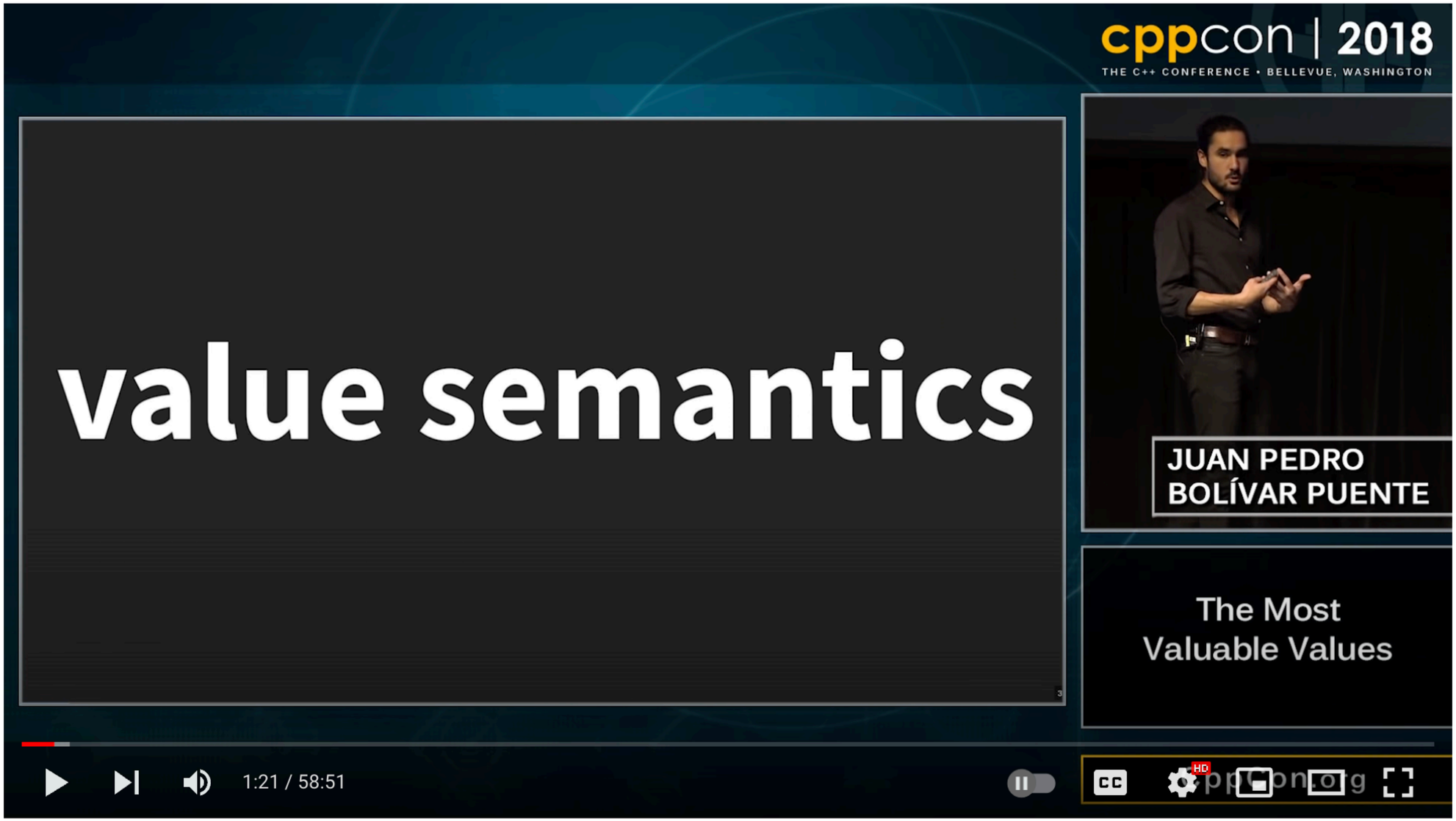
Audio Transcript

Search transcript

Widget uses this cache to respond to queries faster or something. But the cache doesn't actually affect what which it does, except to make it faster. So neither the cache nor its pointer are part of the value, and voila! I just determined the value of my type by identifying its whole part relationships.

youtube.com/watch?v=QthAU-t3PQ4

Values



CppCon 2018: Juan Pedro Bolivar Puente “The Most Valuable Values”

youtube.com/watch?v=_oBx_NbLghY

Most valuable Values

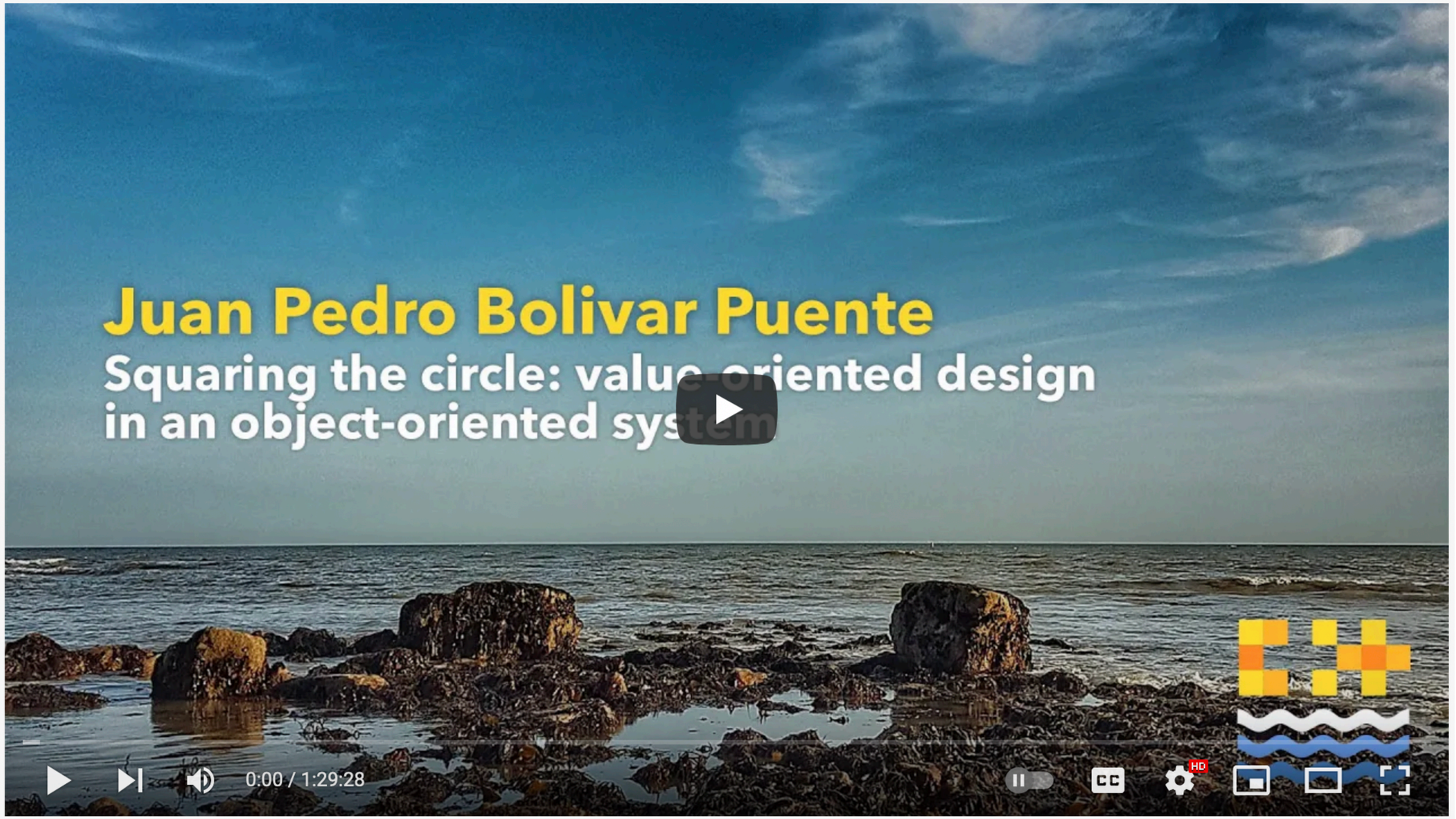
Value-oriented design reconciles functional and procedural programming by focusing on *value semantics*.

Like functional programming, it promotes local reasoning and composition.

It is however *pragmatic* and can be implemented in idiomatic C++,
in existing codebases.

Juan Pedro Bolívar Puente

Values



Value-oriented design in an object-oriented system - Juan Pedro Bolivar Puente [C++ on Sea 2020]

youtube.com/watch?v=SAMR5GJ_GqA

C++ 20 Ranges

The beginning of the end for [begin, end)

Jeff Garland

Ranges

Adaptors

New algorithms

Pipelines

Ranges

Views

Actions*

Lazy evaluation

Projections

Very efficient generated code

A taste of ranges

Print only the **even** elements of a range in **reverse** order:

```
std::for_each(
    crbegin(v), crend(v),
    [](auto const i)
    {
        if(is_even(i))
            cout << i;
    });
```

```
for (auto const i : v
                  | reverse
                  | filter(is_even))
{
    cout << i;
}
```

A taste of ranges

Skip the first **2** elements of the range and print only the **even** numbers of the **next 3** in the range:

```
auto it = cbegin(v);
std::advance(it, 2);
auto ix = 0;
while (it != cend(v) && ix++ < 3)
{
   if (is_even(*it))
      cout << (*it);
   it++;
}
```

```
for (auto const i : v
                  | drop(2)
                  | take(3)
                  | filter(is_even))
{
   cout << i;
}
```

A taste of ranges

Modify an *unsorted* range so that it retains only the **unique** values but in **reverse** order.

```
vector<int> v{ 21, 1, 3, 8, 13, 1, 5, 2 };

std::sort(begin(v), end(v));

v.erase(
   std::unique(begin(v), end(v)),
   end(v));

std::reverse(begin(v), end(v));
```

```
vector<int> v{ 21, 1, 3, 8, 13,
               1, 5, 2 };

v = std::move(v)
    | sort
    | unique
    | reverse;
```

A taste of ranges

Create a range of ***strings*** containing the **last 3** numbers **divisible to 7** in the range **[101, 200]**, in **reverse** order.

```
vector<string> v;

for (int n = 200, count = 0;
     n >= 101 && count < 3; --n)
{
   if (n % 7 == 0)
   {
      v.push_back(to_string(n));
      count++;
   }
}
```

```
auto v = iota_view(101, 201)
       | reverse
       | filter([](auto v) { return v%7==0; })
       | transform(to_string)
       | take(3)
       | to<vector>();
```

It's all about | pipelines

Taking inspiration from Doug McIlroy's UNIX shell script:

```
tr -cs A-Za-z '\n' |
    tr A-Z a-z     |
    sort           |
    uniq -c        |
    sort -rn       |
    sed ${1}q
```

Word frequencies

```
const auto words =
      input_range<string>(std::cin)
    | view::transform(string_to_lower)
    | view::transform(string_only_alnum)
    | view::remove_if(&string::empty)
    | ranges::sort
    | ranges::to<vector>();
```

Word frequencies

```
const auto results = words
  | view::group_by(equal_to())
  | view::transform([] (const auto & grp) {
      const auto size = distance(begin(grp), end(grp));
      const string word = *begin(grp);
      return make_pair(size, word);
    })
  | ranges::sort
  | ranges::to<vector>();
```

Contemporary “Doug McIlroy” examples

Conor Hoekstra's Sushi for 2 🍣

```cpp
template <int N>
constexpr auto sushi_for_two(std::array<int, N> sushi) {
    int current_sushi = 0;
    int sushi_in_a_row = 0;
    int prev_sushi_in_a_row = 0;
    int max_of_mins = 0;
    for (auto const s : sushi) {
        if (current_sushi != s) {
            current_sushi = s;
            if (prev_sushi_in_a_row == 0) {
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row = 1;
            } else {
                auto const min = std::min(sushi_in_a_row, prev_sushi_in_a_row);
                max_of_mins = std::max(max_of_mins, min);
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row = 1;
            }
        } else {
            sushi_in_a_row += 1;
        }
    }
    auto const min = std::min(sushi_in_a_row, prev_sushi_in_a_row);
    max_of_mins = std::max(max_of_mins, min);
    return max_of_mins * 2;
}
```


```cpp
auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(_eq_)
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```

Tristan Brindle
@tristanbrindle

github.com/tcbrindle/flux

```cpp
auto sushi_for_two(std::vector<int> const& sushi)
{
    return 2 * flux::ref(sushi)
                .chunk_by(std::equal_to{})
                .map(flux::count)
                .pairwise_map(std::ranges::min)
                .max()
                .value();
}
```

godbolt.org/z/oras9sEE3

Both are here

Wednesday, 11am

Iteration Revisited

cppnorth.digital-medium.co.uk/session/iteration-revisited/

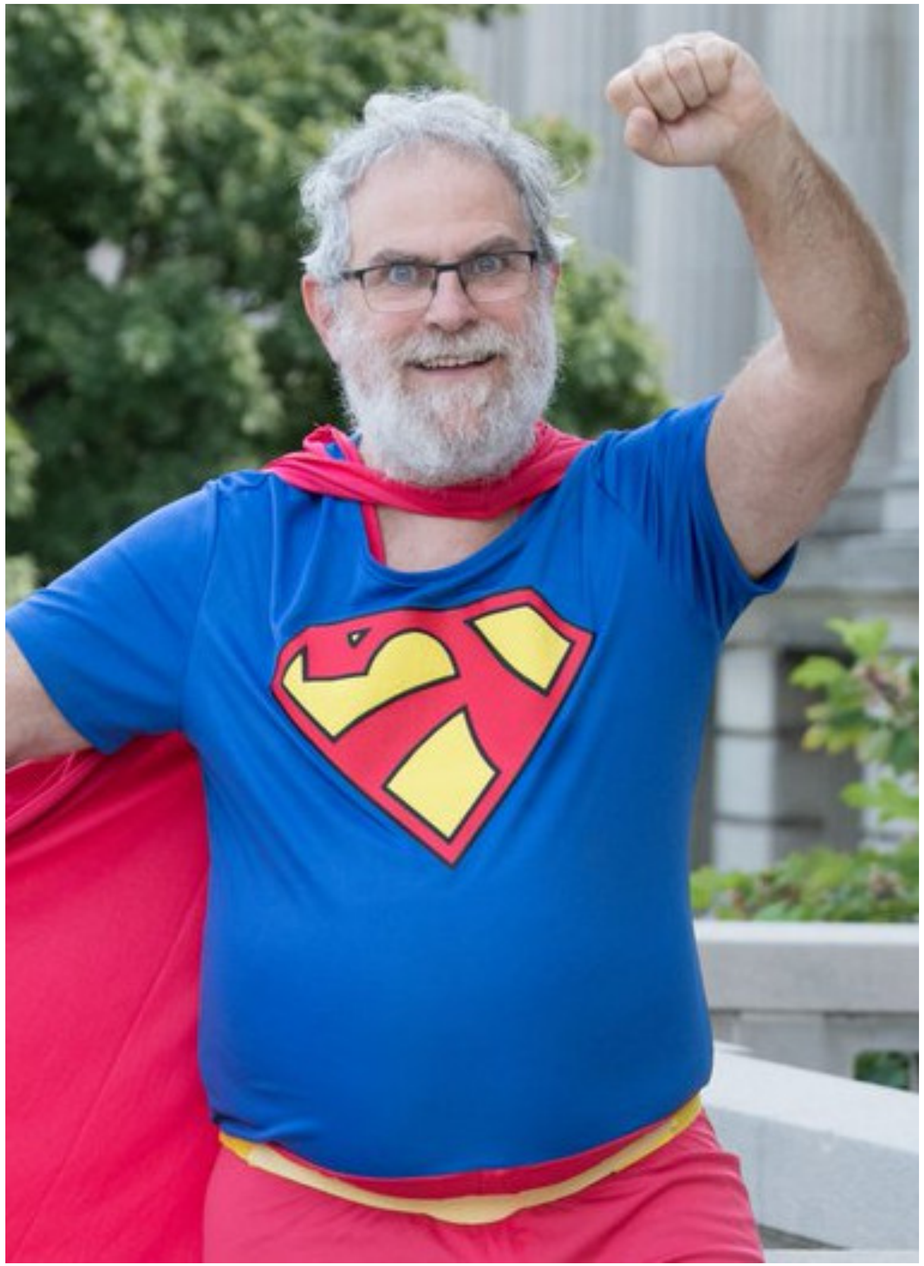
Cpp North

Wednesday, 1:30pm

New Algorithms in C++23

cppnorth.digital-medium.co.uk/session/new-algorithms-in-c23/

Phil Wadler developed type classes and monads, two of the main innovations of Haskell



<*> Čukić

"Make your code readable.

Pretend the next person who looks at your code is a psychopath and they know where you live."

Phil Wadler

And Then() Some(T)

July 2023

Victor Ciura
Principal Engineer
Visual C++

@ciura_victor
@ciura_victor@hachyderm.io