The Imperatives Must Go!

Meeting C++

November 2022

@ciura_victor

Victor Ciura
Senior SW Engineer
Visual C++

Abstract

Can a language whose official motto is “Avoid Success at All Costs” teach us new tricks in modern C++ ?

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

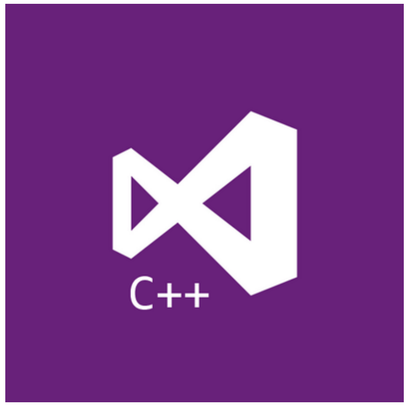
Indeed, modern C++ has become more functional. From mundane concepts like lambdas & closures, std::function, values types and constants, to composability of STL algorithms, lazy ranges, folding, mapping or even higher-order functions in STL. Did I mention Rust yet?

In this session we’ll analyze a bunch of FP techniques in C++ and see how they help make our code shorter, clearer and faster, by embracing a declarative vs. an imperative style. We’ll visit the functional parts of current STL, use algebraic data types (ADT) and learn about the new FP stuff coming in the next C++ standard, like ranges or monadic extensions to std::future, std::optional and std::expected. Brace yourselves for a bumpy ride including composition, lifting, currying, partial application, pure functions, maybe even pattern matching and lazy evaluation.

About me

Advanced Installer

Clang Power Tools



Visual C++

@ciura_victor

1/n

This is meant as an introductory presentation to the concepts to follow.

Depending on how this lands, sequels will cover some of these topics in depth.

Don't worry, there are no cliffhangers...

🔥 Hot take typing

If it looks like a hot take, if it feels like a hot take... it probably is 😈

FP in 10

Functional Programming

What is it all about ?

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

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

A Crash Course in Category Theory - Bartosz Milewski https://www.youtube.com/watch?v=JH_Ou17_zyU

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

Is it easier to think like a machine than to do math?

A Crash Course in Category Theory - Bartosz Milewski https://www.youtube.com/watch?v=JH_Ou17_zyU

Semantics

- The meaning of a program
- Operational semantics: local, progress oriented
  - Execute program on an abstract machine in your brain
- Denotational semantics
  - Translate program to math
- Math: an ancient language developed for humans

A Crash Course in Category Theory - Bartosz Milewski https://www.youtube.com/watch?v=JH_Ou17_zyU

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***

Let's address the in the room...

Let's address the in the room...

Haskell

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

What do you mean ?

Summing the integers 1 to 10 in C++/Java/C#

```
int total = 0;
for (int i = 1; i ≤ 10; i++)
   total = total + i;
```

The computation method is variable assignment.

Summing the integers 1 to 10 in Haskell

```
sum [1..10]
```

The computation method is function application.

Functional

Non-Functional

WHAT

HOW

A SOLID summary:

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

235 121

wikipedia.org/wiki/SOLID

Historical Background

Historical Background

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

Historical Background

1930s

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

Historical Background

1950s

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

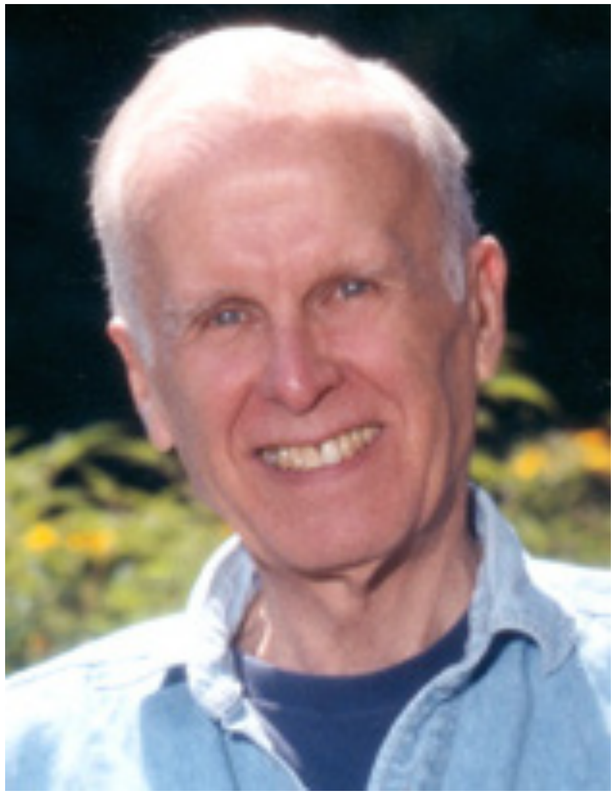
Historical Background

1960s

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

Historical Background

1970s



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

Historical Background

1970s

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

Historical Background

1970-80s

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

Historical Background

1987

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

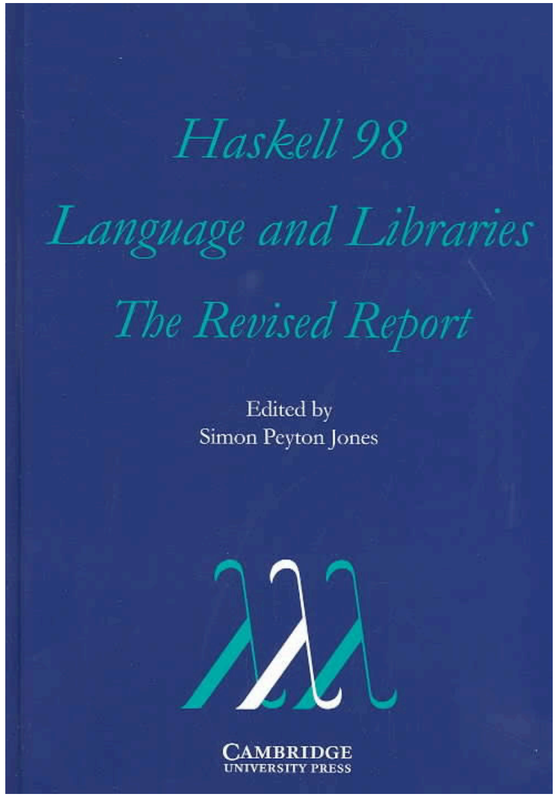
Historical Background

1990s

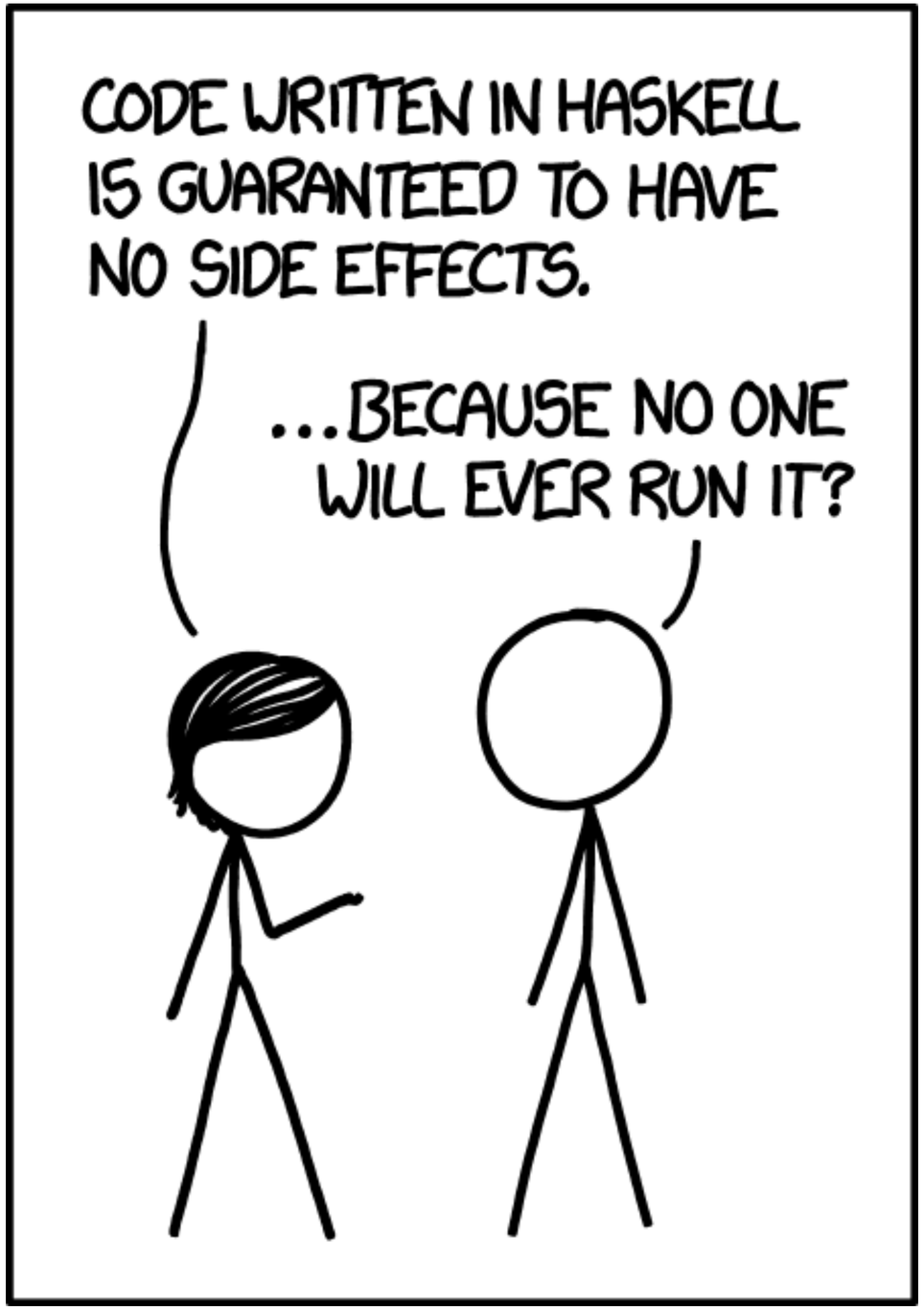
Phil Wadler and others develop type classes and monads,
two of the main innovations of Haskell

Historical Background

2003
2010



The committee publishes the Haskell Report, defining a stable version of the language; an updated version was published in 2010



xkcd.com/1312/

Why (not) Haskell ?

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?

Why (not) Haskell ?

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

My claim is that it has.

Why (not) Haskell ?

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

My claim is that it has.

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.

Why (not) Haskell ?

Why (not) Haskell ?

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

Why (not) Haskell ?

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

Why (not) Haskell ?

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.

Why (not) Haskell ?

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.

Why (not) Haskell ?

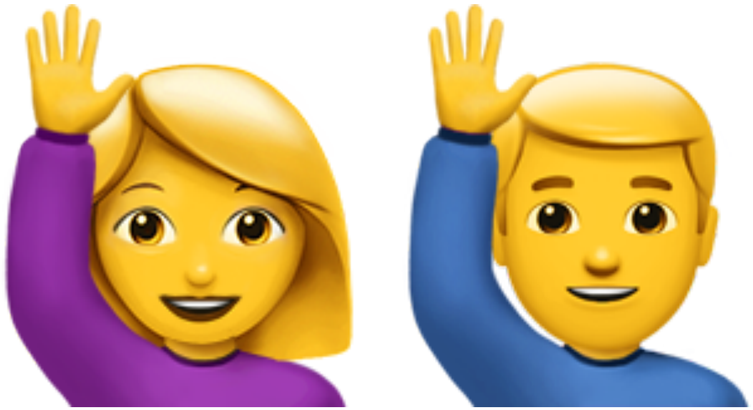
Functional Programming ideas that have been around for over 40 years are rediscovered to solve our current software complexity problems.

Indeed, contemporary C++ has become more functional.

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

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]
```

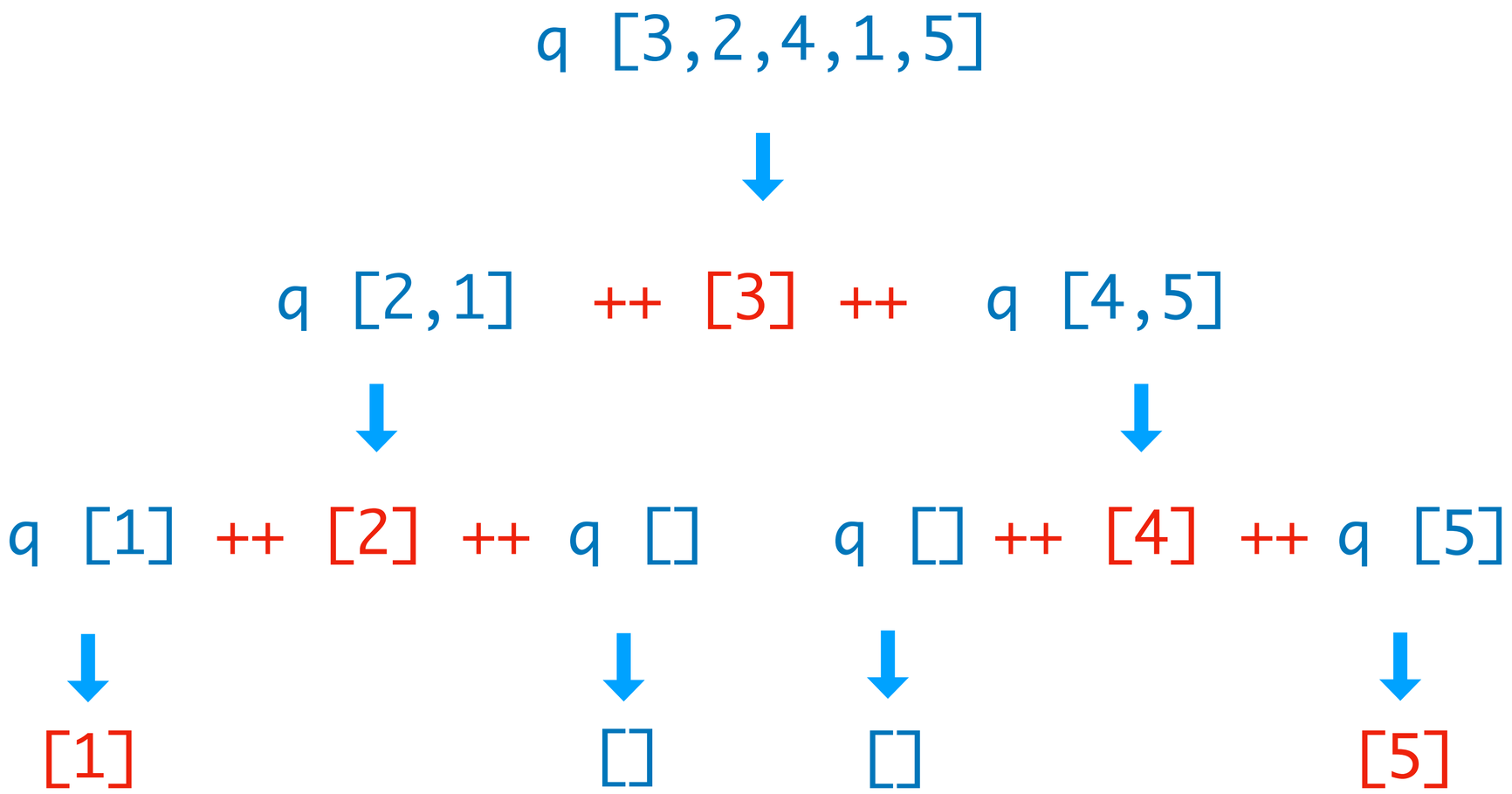


What does `f` do ?

Quick Sort

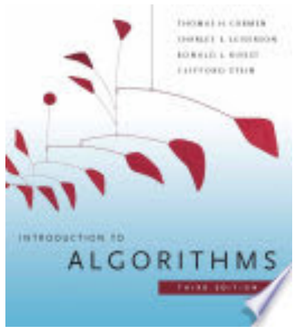
```
qsort :: Ord a ⇒ [a] → [a]
qsort []     = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

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

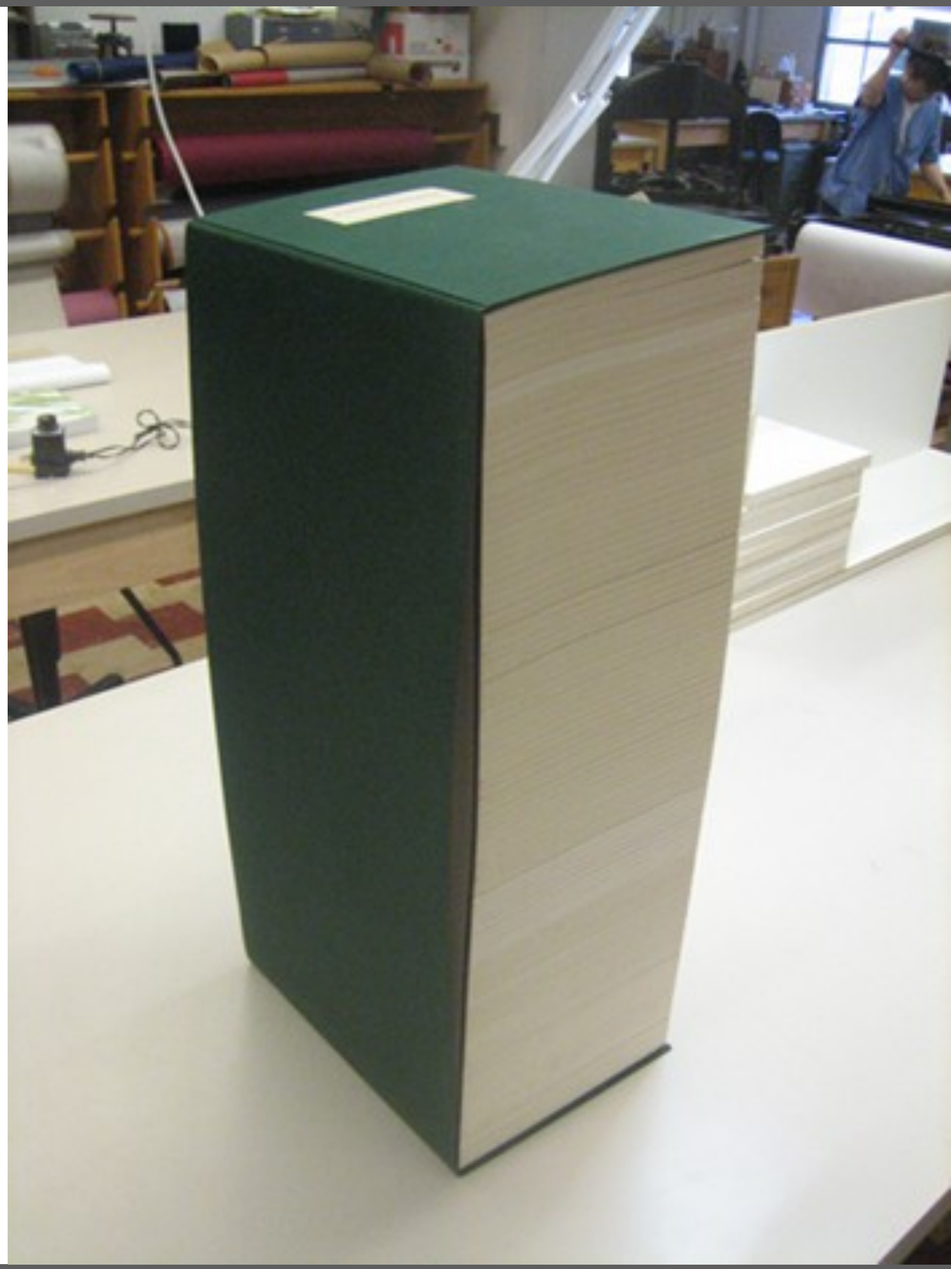
FP...

How do I start on this journey?

Prerequisites

Category Theory
for Programmers

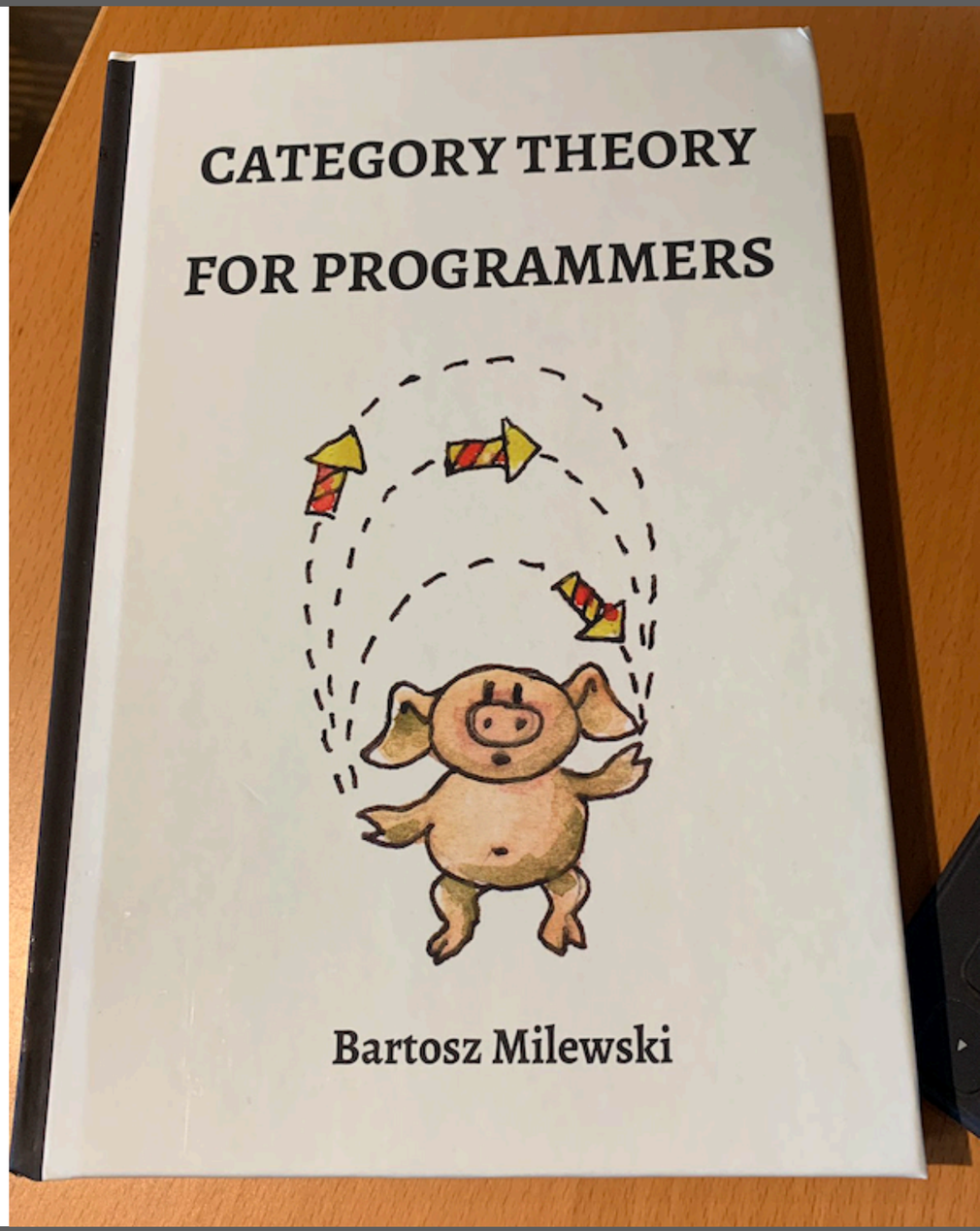
😅



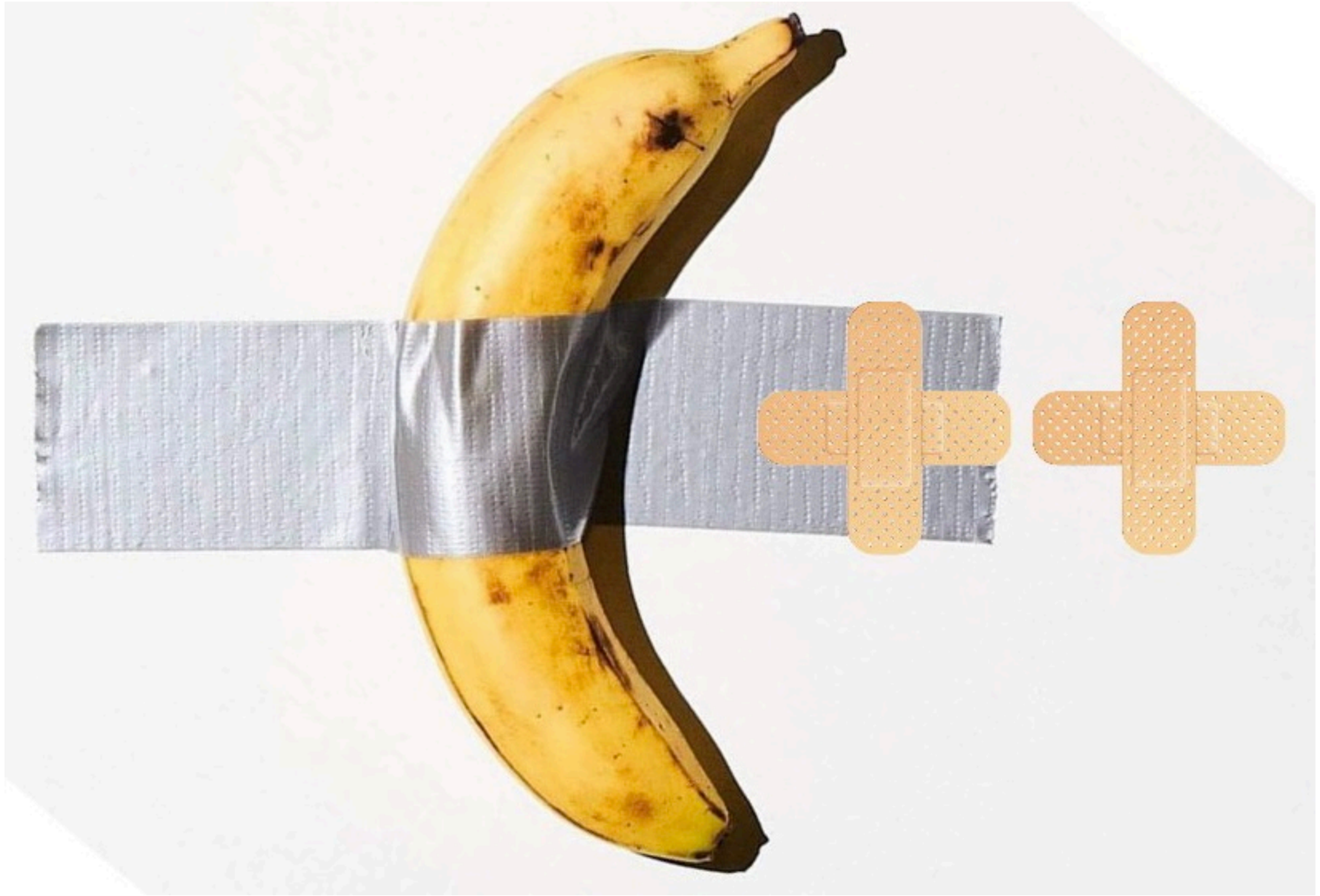
The Book

Bartosz Milewski
@BartoszMilewski

github.com/hmemcpy/milewski-ctfp-pdf



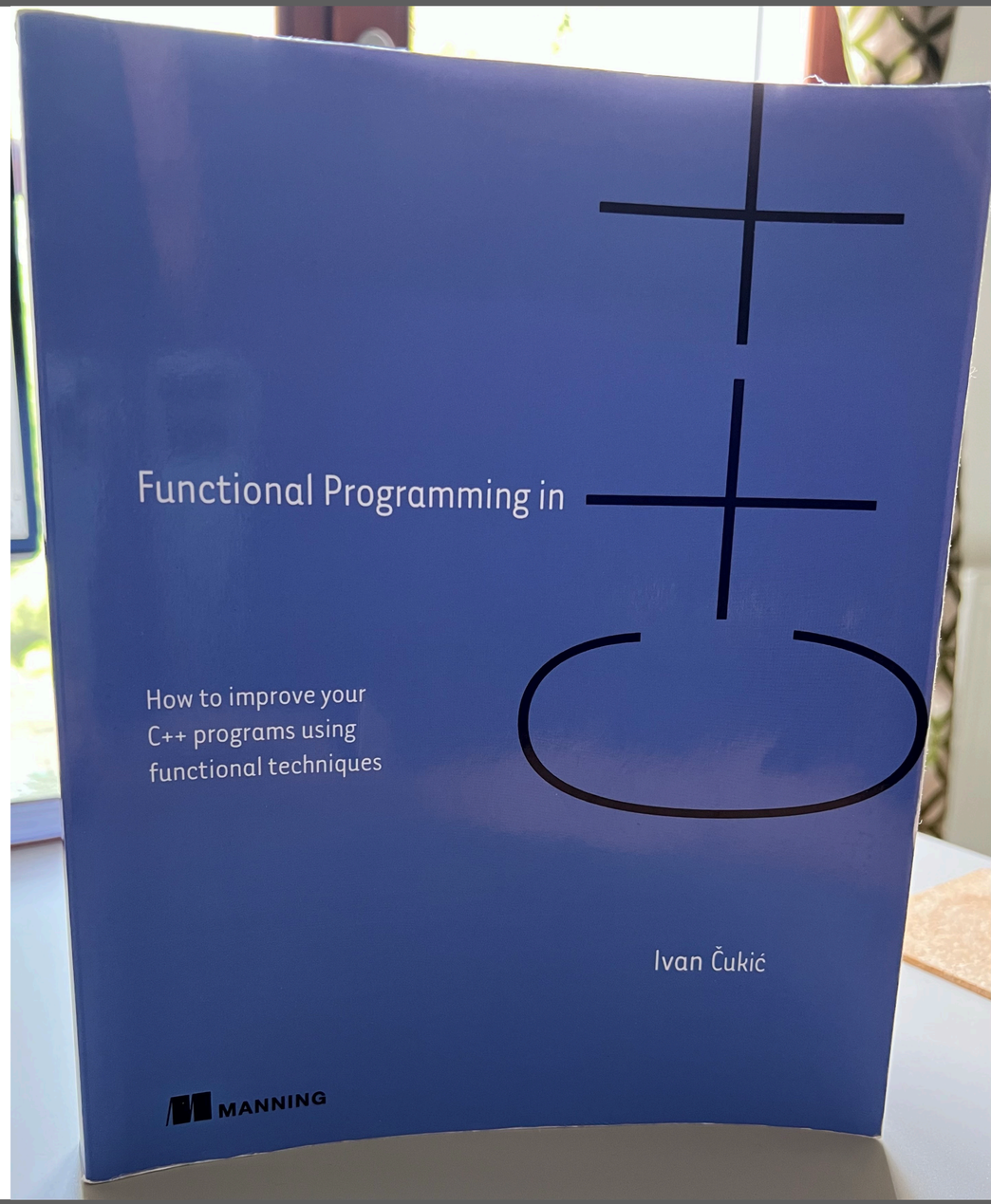
C++



twitter.com/tvaneerd/status/1387

The Book

Ivan Čukić
@ivan_cukic



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

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.

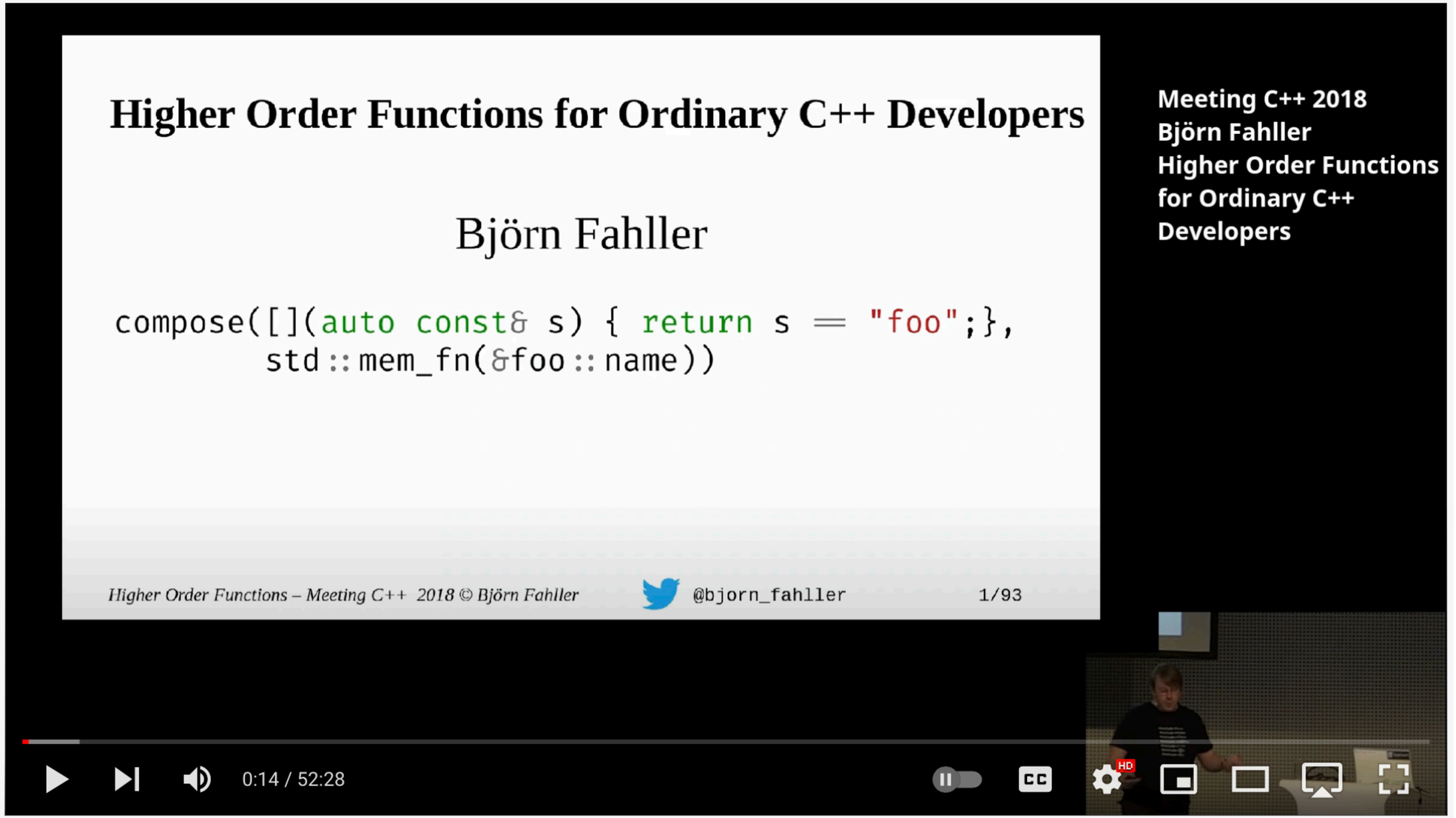
```
#define LIFT_THRICE(...)                    \
       noexcept(noexcept(__VA_ARGS__))      \
        -> decltype(__VA_ARGS__)            \
        {                                   \
          return __VA_ARGS__;               \
        }

#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
```

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:

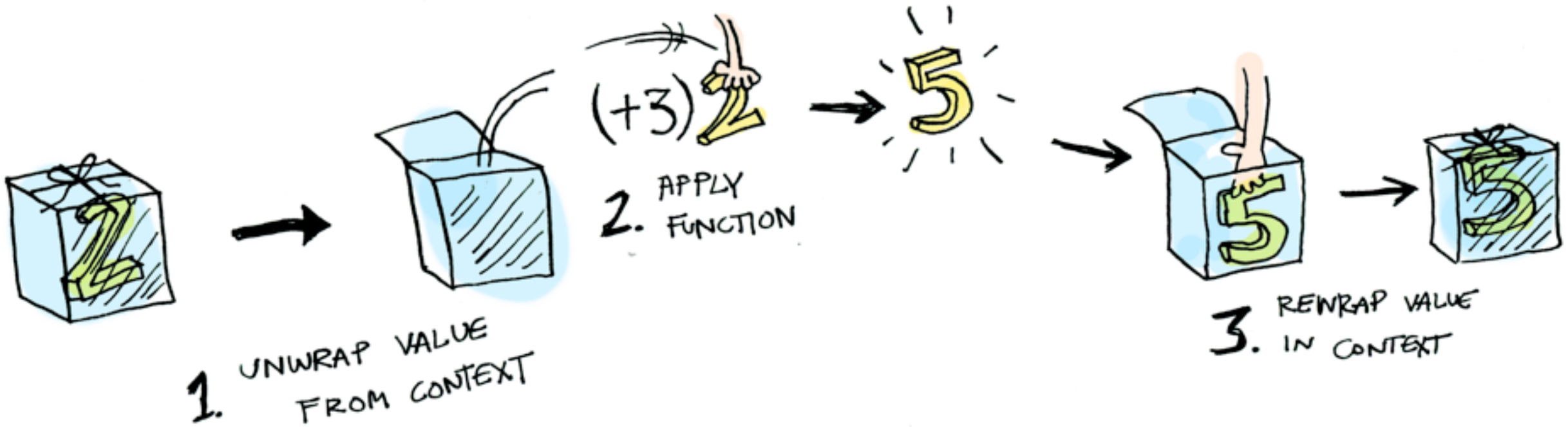
- `unique_ptr<T> p;`
- `shared_ptr<T> p;`
- `vector<T> v;`
- `optional<T> o;`
- `function<T(int)> f;`

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.



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

📦 The Box

`std::optional` can simplify APIs

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

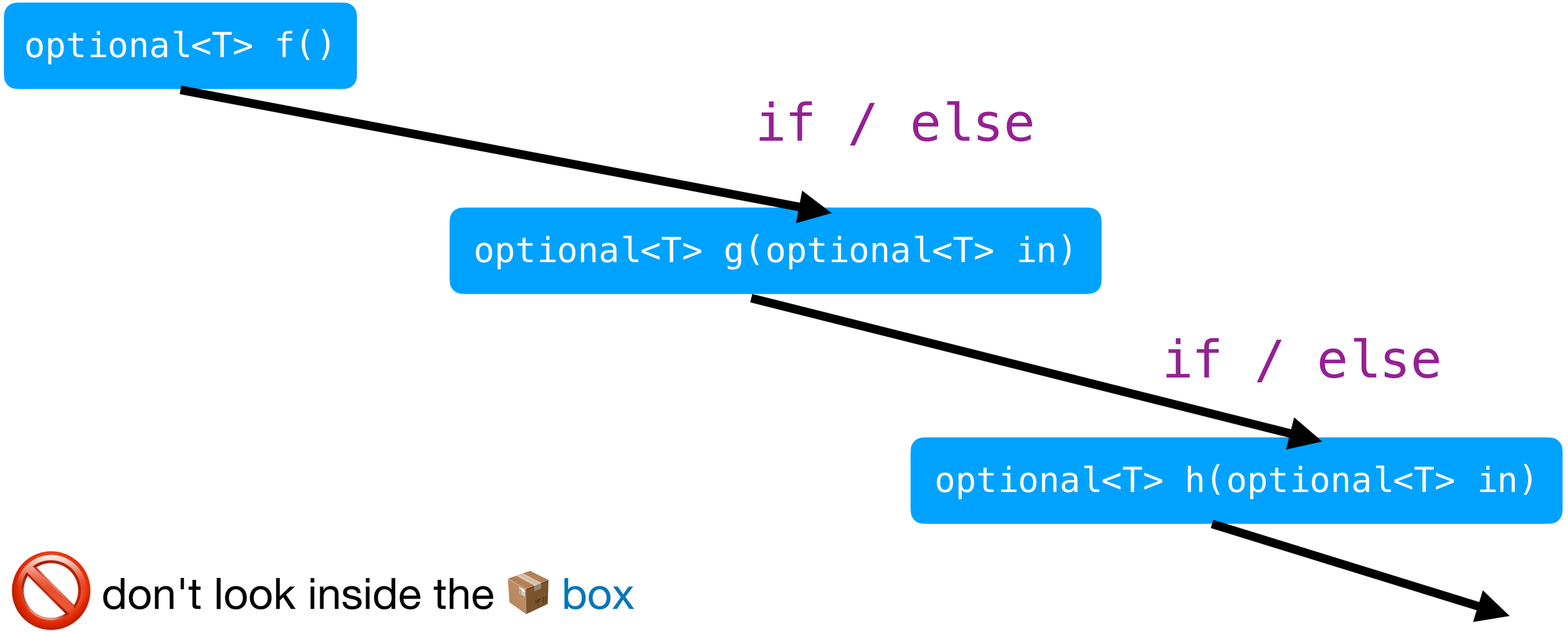
  Haskell (`Maybe`) or Rust (`Option<T>`)

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

📦 The Box

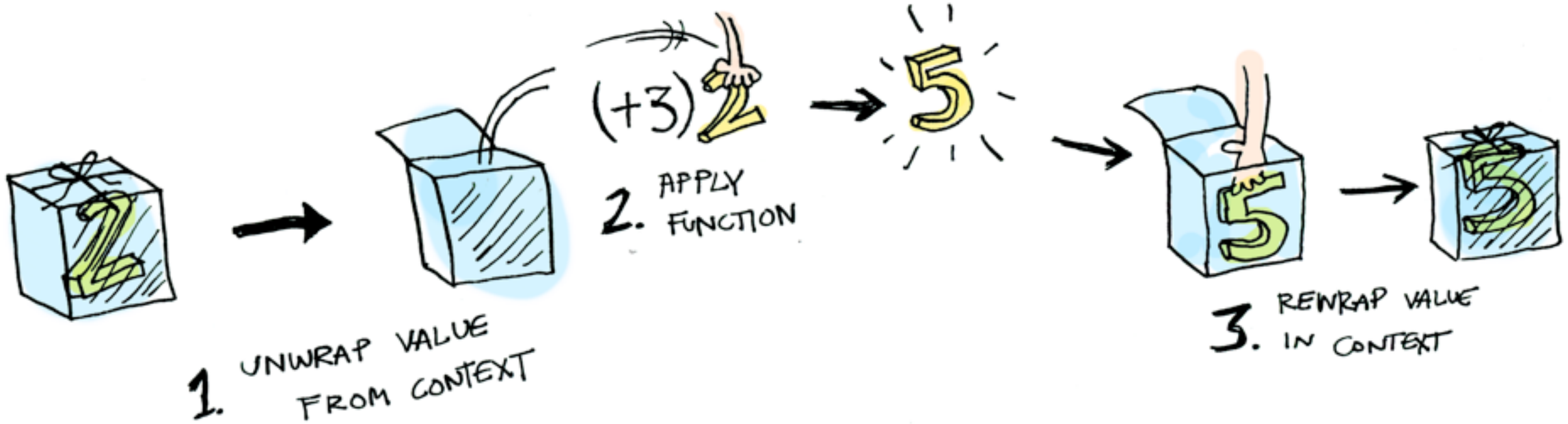
doc.rust-lang.org/rust-by-example/error/option_unwrap

📦 The Box



🚫 don't look inside the 📦 box

📦 The Box



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.

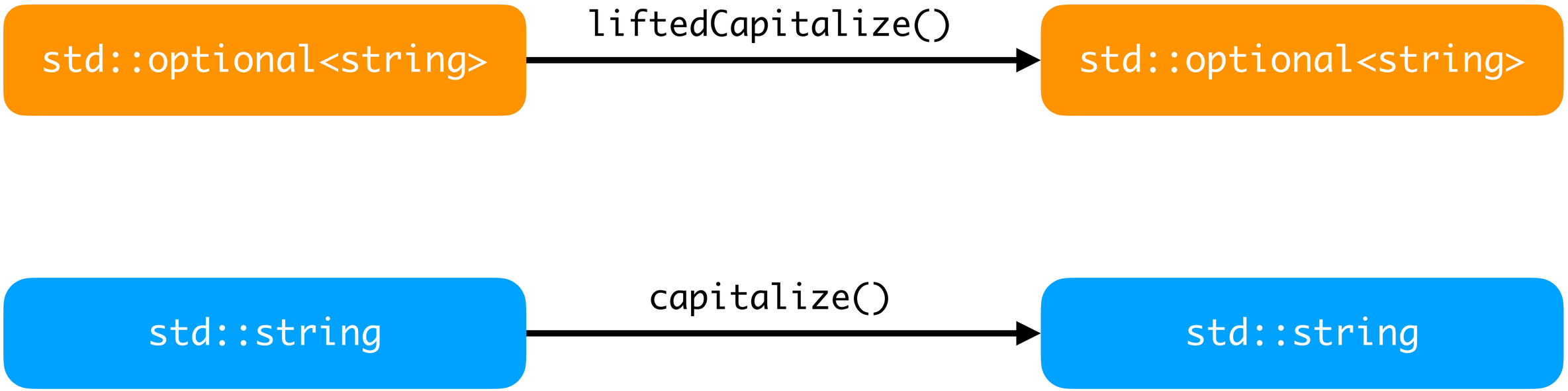
```
string capitalize(string str);
...

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()`

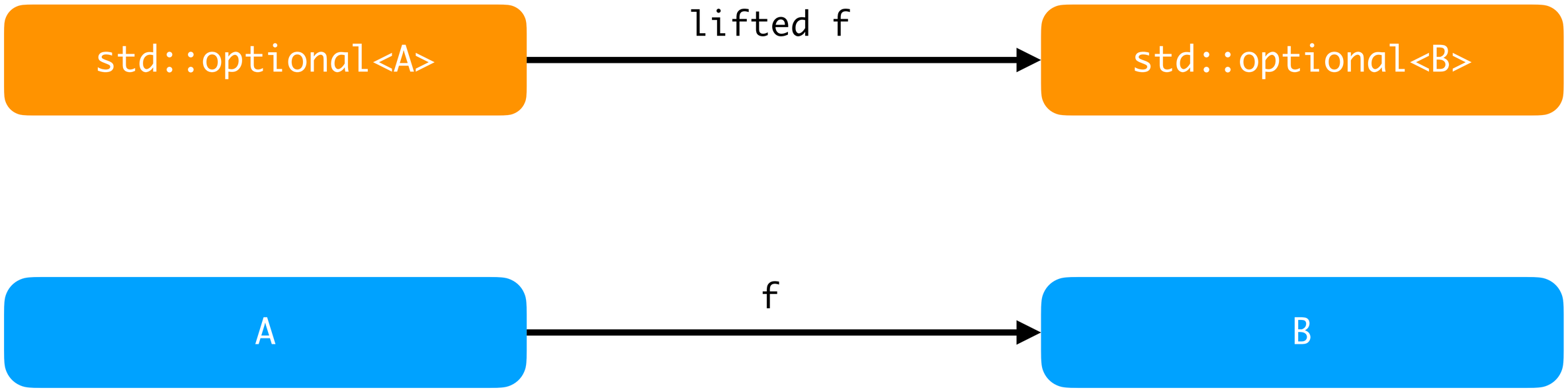


Lifting any function

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;
}
```

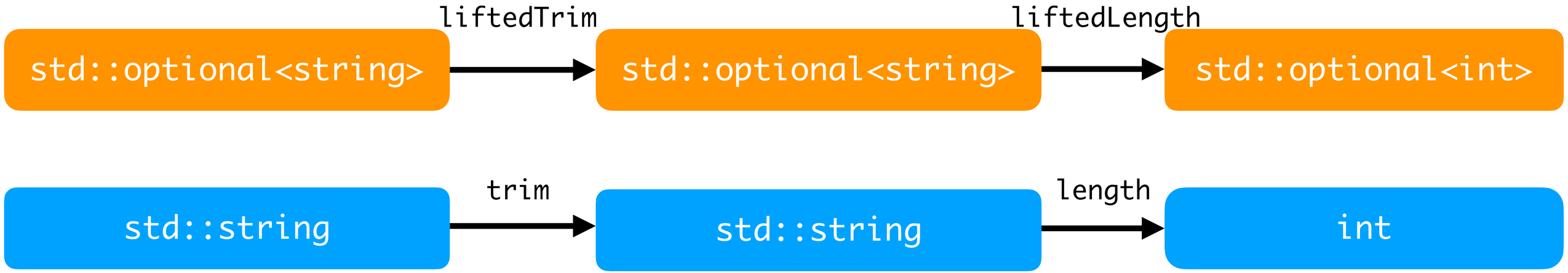
Lifting any function



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));
```



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
}
```

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:

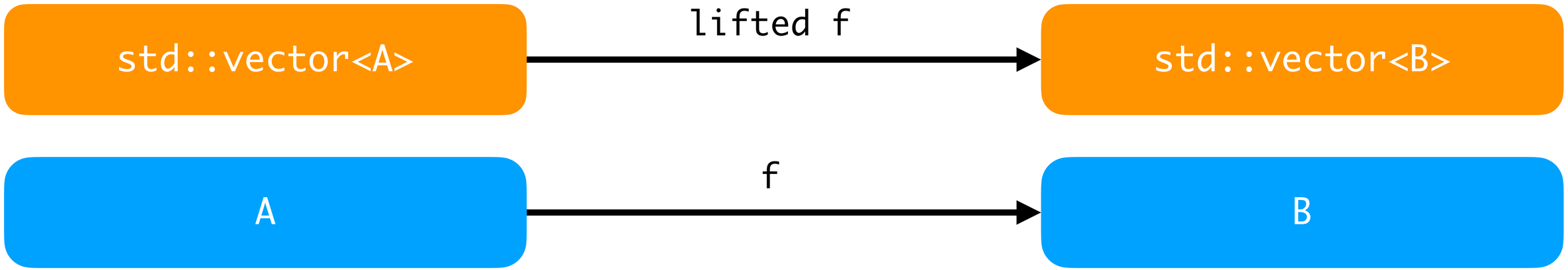
```
optional<int64_t> current_pc = ... ; // function address
...

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

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;
  std::transform(v.begin(), v.end(), back_inserter(result), f);
  return result;
}
```

higher-order function



Lifting a function to a vector

Lifted `length` operates on `vector<string>` and produces `vector<int>`

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

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

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)

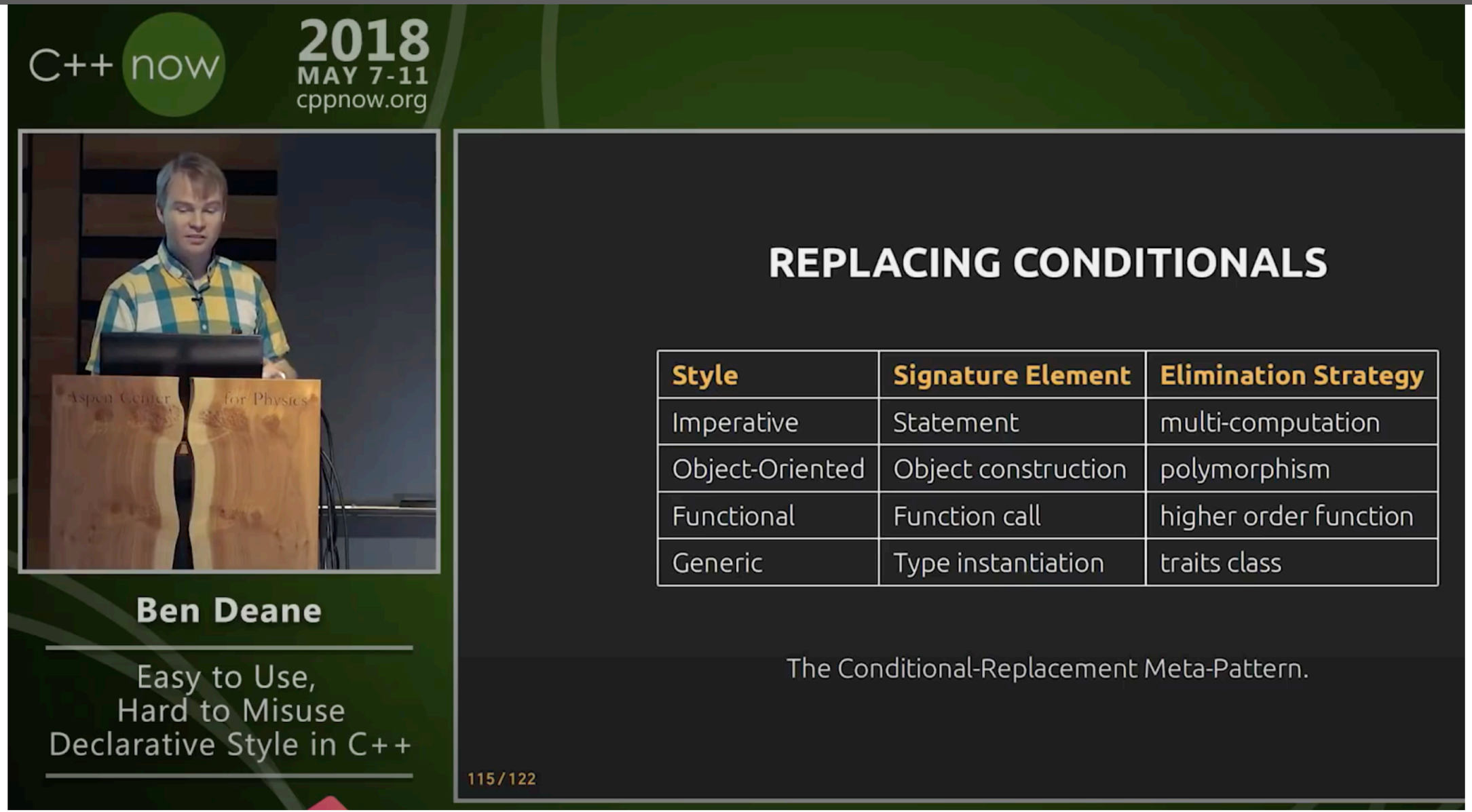
```
optional<int> string_view_to_int(string_view sv)
{
  const auto first = sv.data();
  const auto last  = first + sv.size();

  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()
```

Declarative style



youtube.com/watch?v=2ouxETt75R4

Values

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

Related session

Values

Safety, Regularity, Independence, Projection, and the Future of Programming

15:15 - 16:15 Wednesday 14th September 2022 MDT Summit 2 & 3 / Online D

Beginner | Intermediate | Advanced | Expert | Value Semantics

+ Add to Schedule

Support for first-class user-defined value types may be among C++'s greatest strengths—one that most recent language designs have sadly failed to emulate. That said, although value types are everywhere in C++, we don't have a commonly accepted definition of “value semantics”, and we tend to use the phrase with only an intuitive idea of what it means. This talk offers a deeper understanding of value semantics, defining it in a way that in turn reveals surprising truths about programming in general. We'll expose the value semantics that underlies our mental model even when we're “forced” to use pointers or references, and discuss how a future C++ might close that expressivity gap, improving safety, performance, and programmer confidence. We'll conclude with some guidelines you can use today to improve your programs, and propose the next must-see session for value semantics lovers.

This presentation lays groundwork for another talk, “Val wants to be your friend.” If you're interested in that talk, you'll want to see this one first.

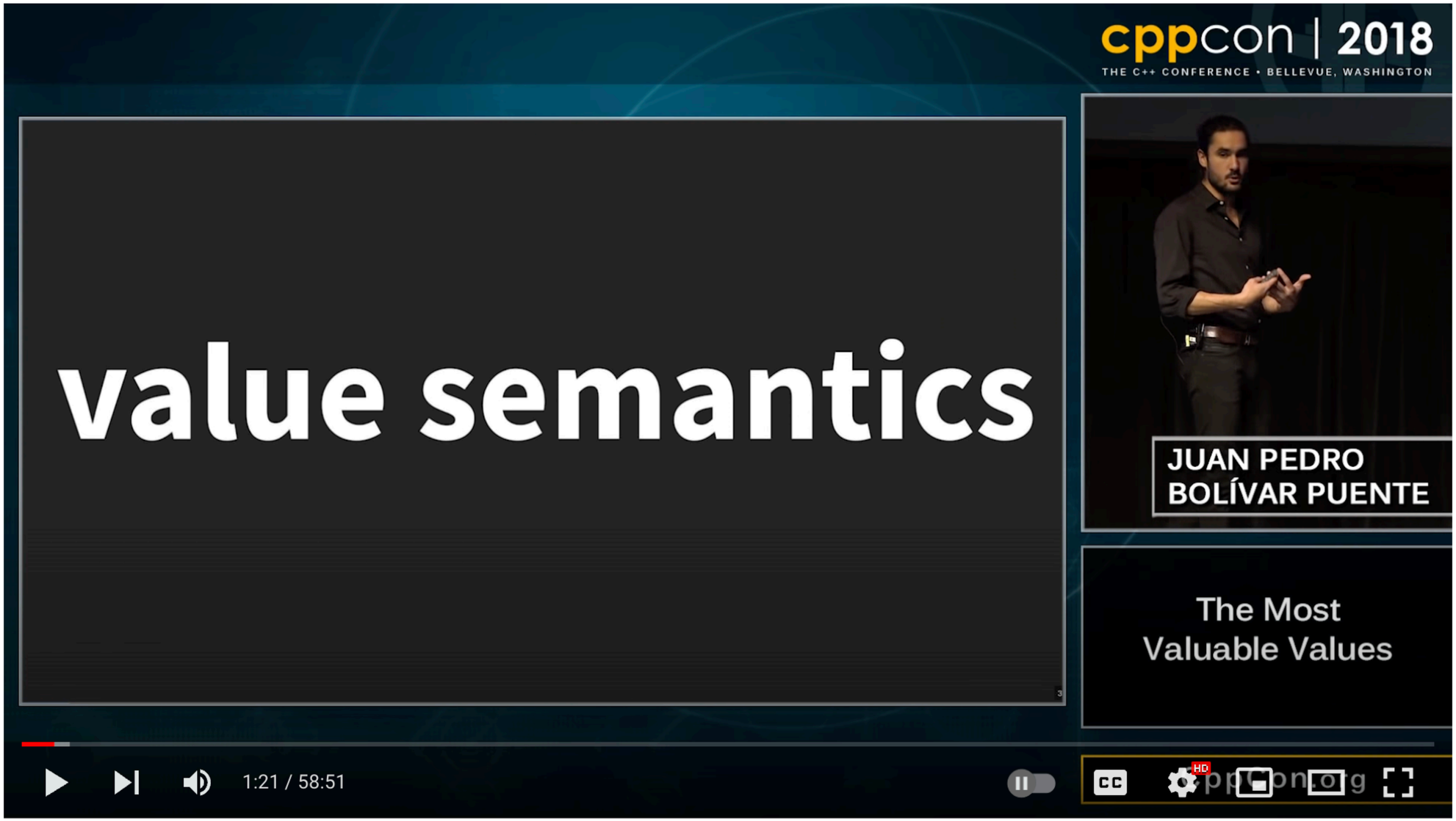
Dave Abrahams

Principal Scientist
Adobe

Dave Abrahams is a founding contributor of the Boost C++ Libraries project and the founder of the first annual C++ conference, BoostCon/C++Now. He is a contributor to the C++ standard, and was a principal designer of the Swift programming language. He recently spent seven years at Apple, culminating in the creation of the declarative SwiftUI framework, worked at Google on the Swift for TensorFlow project and, briefly, on the Carbon language, and is now a principal scientist at Adobe's Software Technology Lab.

cppcon.digital-medium.co.uk/session/2022/values/

Values



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

youtube.com/watch?v=_oBx_NbLghY

Most valuable Values

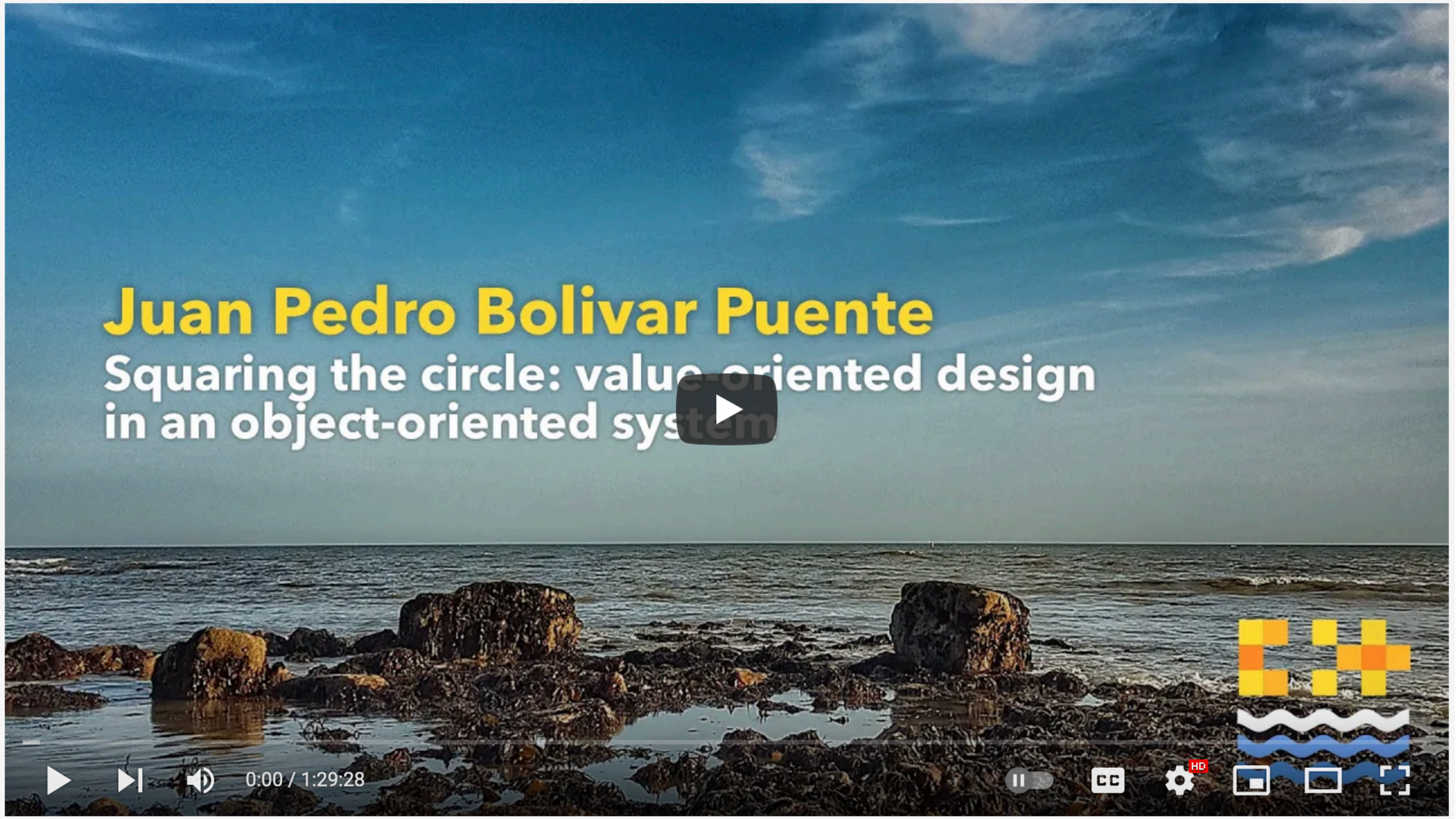
Value-oriented design reconciles functional and procedural programming by focusing on *value semantics*.

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

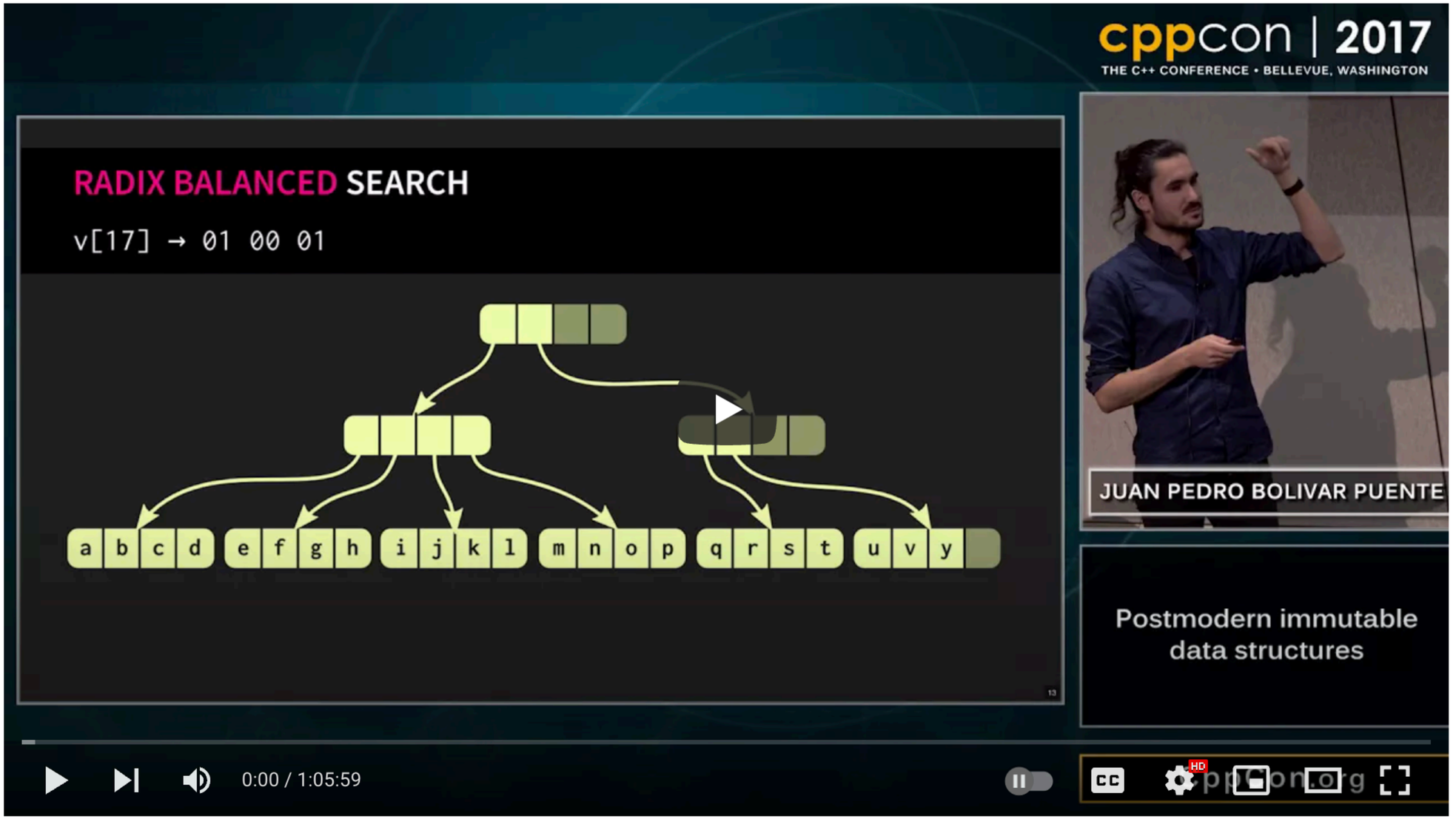
Values



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

youtube.com/watch?v=SAMR5GJ_GqA

Immutable DS



CppCon 2017

CppCon 2017: Juan Pedro Bolivar Puente "Postmodern immutable data structures"

youtube.com/watch?v=sPhpeIUfu8Q

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>();
```

A taste of ranges

C++20 ranges ruined one more interview question 😄

```
auto strings = std::string_view{"Hello C++ 20"}
                  | std::views::split(' ');
```

a range of ranges

lazily evaluated

```
for (const auto & ref : strings)
  std::cout << std::string_view{ref.begin(), ref.end()} << '\n';
```

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>();
```

Word frequencies

```
for (auto value : results | view::reverse
                          | view::take(n))
{
  cout << value.first << ": " << value.second << "\n";
}
```

Gotchas with ranges

C++20 ranges library is fantastic tool, but watch out for gotchas ⚠️

- views have *reference* semantics => all the reference gotchas apply
- as always with C++, **const** is shallow and doesn't propagate (as you might expect)
- some functions do caching, eg. `.begin(), .empty() | filter | drop`
- don't hold on to views or try to reuse them
  - safest to use them ad-hoc, as temporaries
  - if needed, better "copy" them (cheap) for reuse

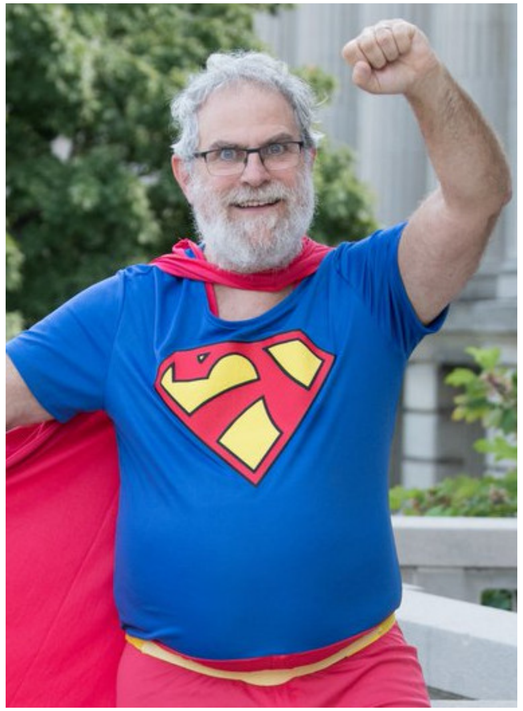
Remember him?

1990s

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

<*> Čukić

Takeaway



"Make your code readable.

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

Phil Wadler

The Imperatives Must Go!

Meeting C++

November 2022

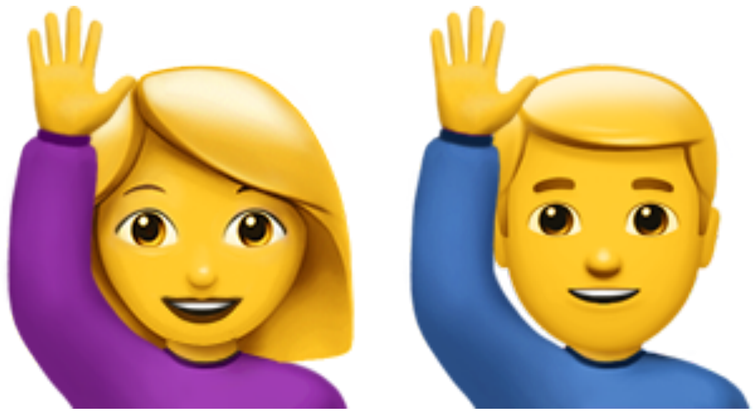
@ciura_victor

Victor Ciura
Senior SW Engineer
Visual C++

Bonus problem

Counting adjacent repeated values in a sequence.

How many of you solved this textbook exercise before ?
(in any programming language)



Counting adjacent repeated values in a sequence

{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }

Counting adjacent repeated values in a sequence

Visual hint:

```
{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }
```

Counting adjacent repeated values in a sequence

Visual hint:

```
{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }
```


```
{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }
```

Counting adjacent repeated values in a sequence

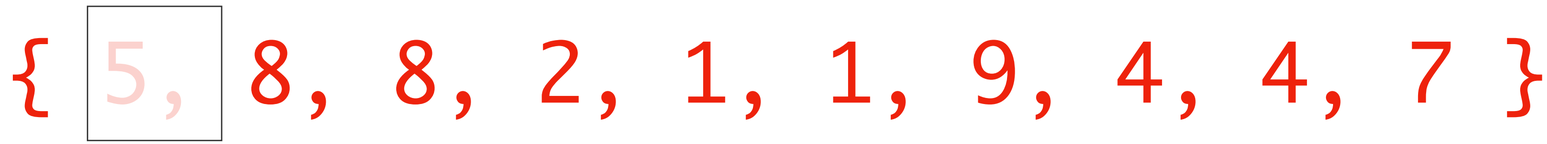
Visual hint:

```
{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }
```


```
{ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 }
```

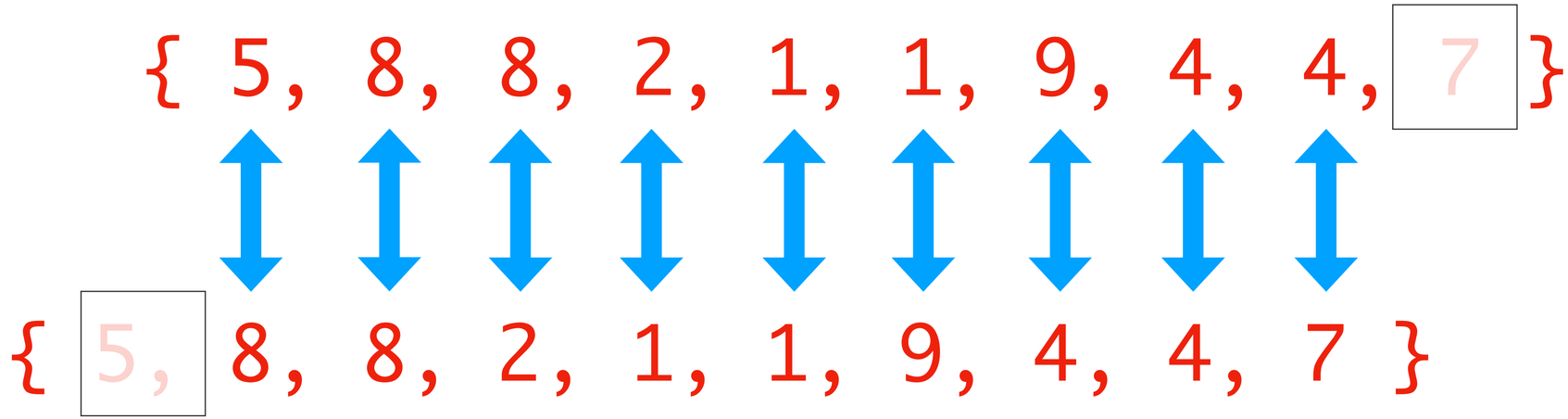
Counting adjacent repeated values in a sequence

Visual hint:



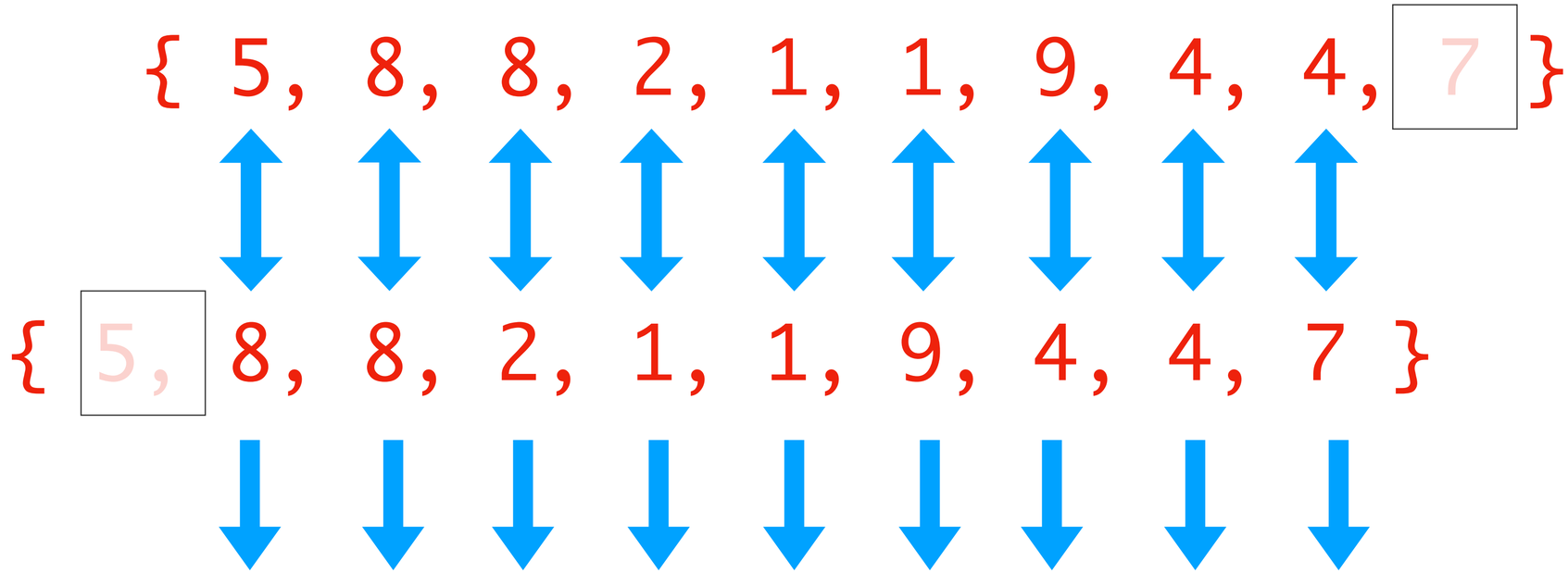
Counting adjacent repeated values in a sequence

Visual hint:



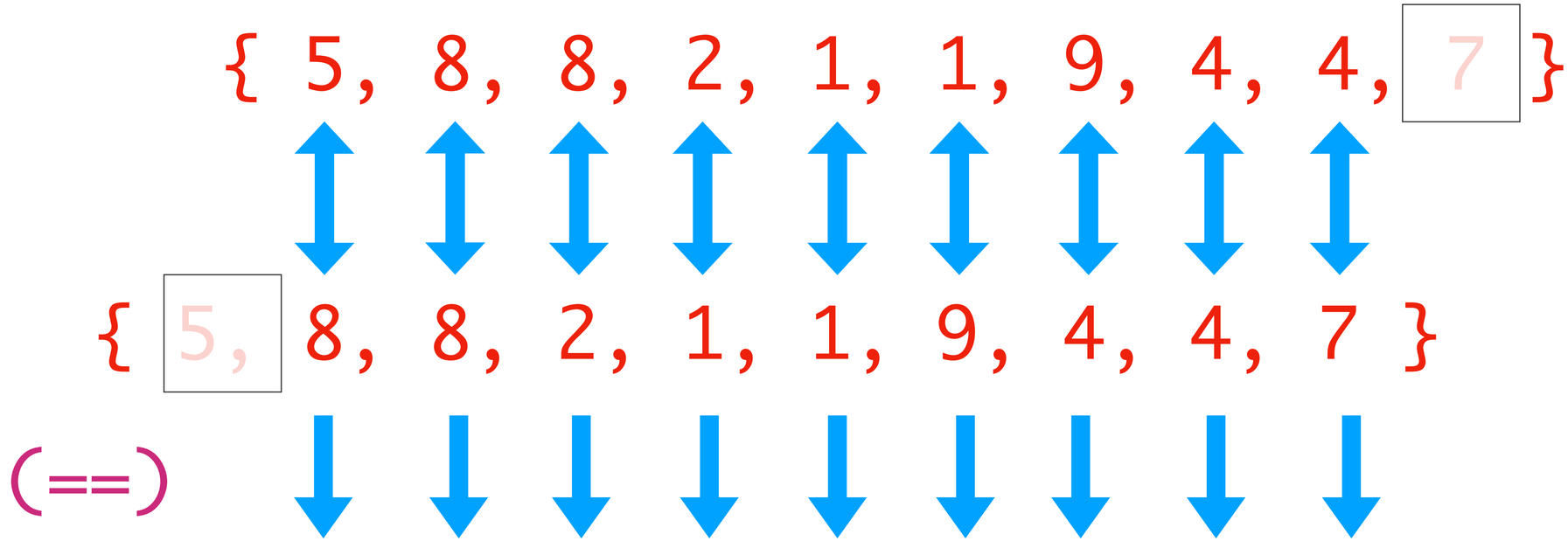
Counting adjacent repeated values in a sequence

Visual hint:



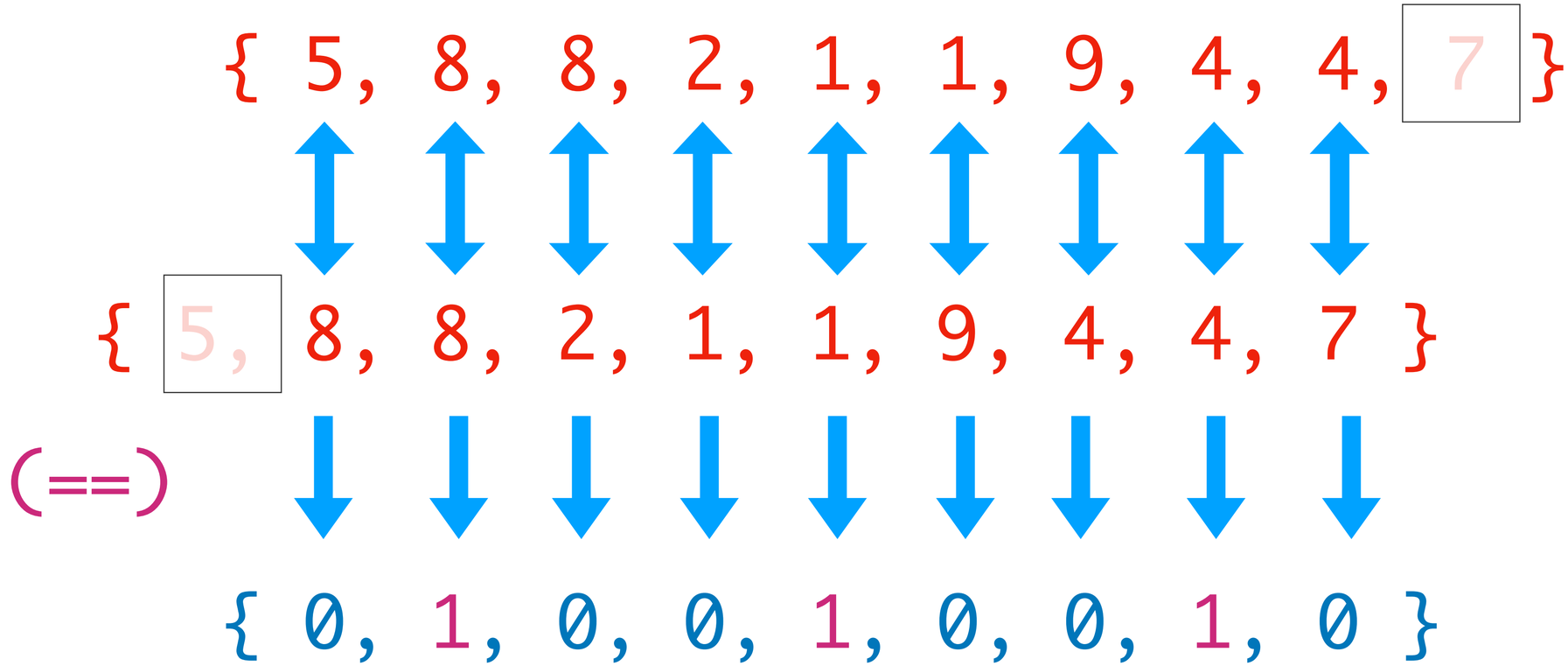
Counting adjacent repeated values in a sequence

Visual hint:



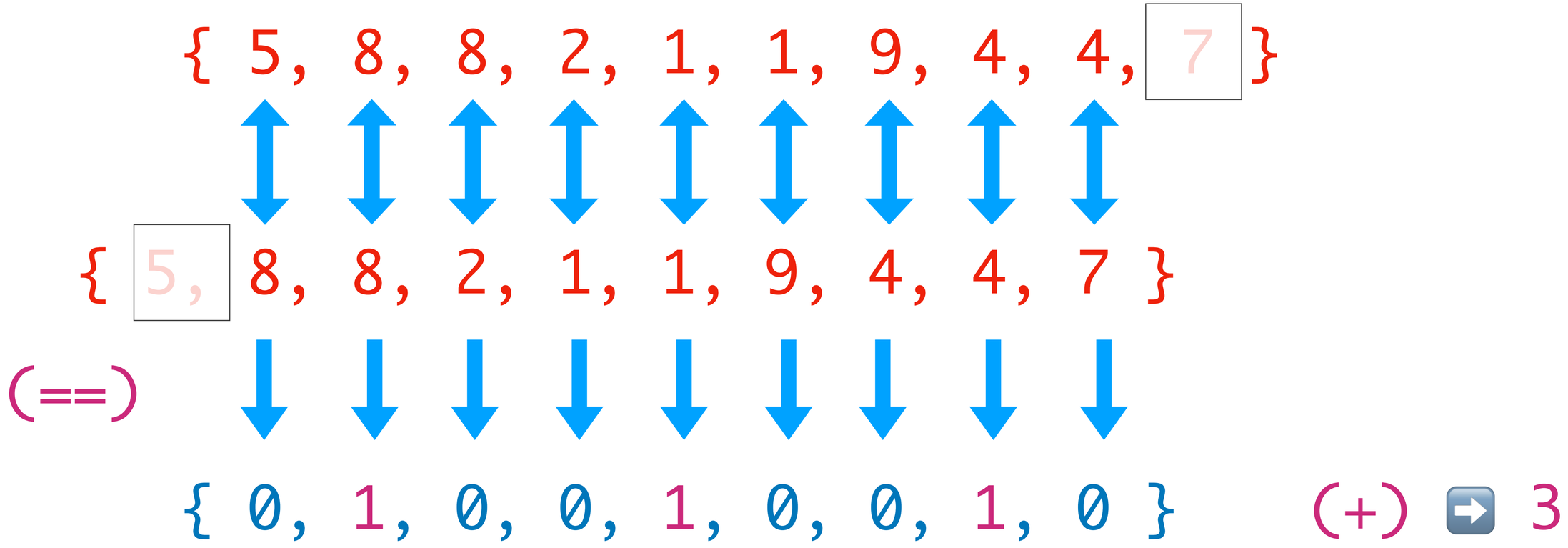
Counting adjacent repeated values in a sequence

Visual hint:



Counting adjacent repeated values in a sequence

Visual hint:



C++ Counting adjacent repeated values in a sequence

Let me guess... a bunch of `for` loops, right ?

C++ Counting adjacent repeated values in a sequence

Let me guess... a bunch of `for` loops, right ?

How about something shorter ?

An STL `algorithm` maybe ?

C++ Counting adjacent repeated values in a sequence

```
template<class InputIt1, class InputIt2,
         class T,
         class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T init,
                BinaryOperation1 op1  // "sum" function
                BinaryOperation2 op2) // "product" function
{
  while (first1 != last1)
  {
      init = op1(init, op2(*first1, *first2));
      ++first1;
      ++first2;
  }
  return init;
}
```

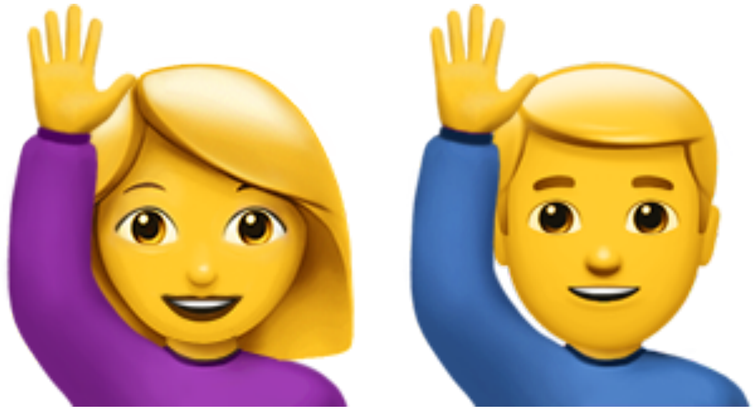
https://en.cppreference.com/w/cpp/algorithm/inner_product

C++ Counting adjacent repeated values in a sequence

```
template <typename T>
int count_adj_equals(const T & xs) // requires non-empty range
{
  return std::inner_product(
    std::cbegin(xs), std::cend(xs) - 1, // to penultimate elem
    std::cbegin(xs) + 1,                // collection tail
    0,
    std::plus{},
    std::equal_to{}); // yields boolean => 0 or 1
}
```

C++ Counting adjacent repeated values in a sequence

If you found that piece of code in a code-base,
would you **understand** what it does* ?



* without my cool diagram & animation

Counting adjacent repeated values in a sequence

Let's go back to Haskell for a few minutes...

Counting adjacent repeated values in a sequence

Visual hint:

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```

Counting adjacent repeated values in a sequence

Visual hint:

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```


```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```

Counting adjacent repeated values in a sequence

Visual hint:

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```

Counting adjacent repeated values in a sequence

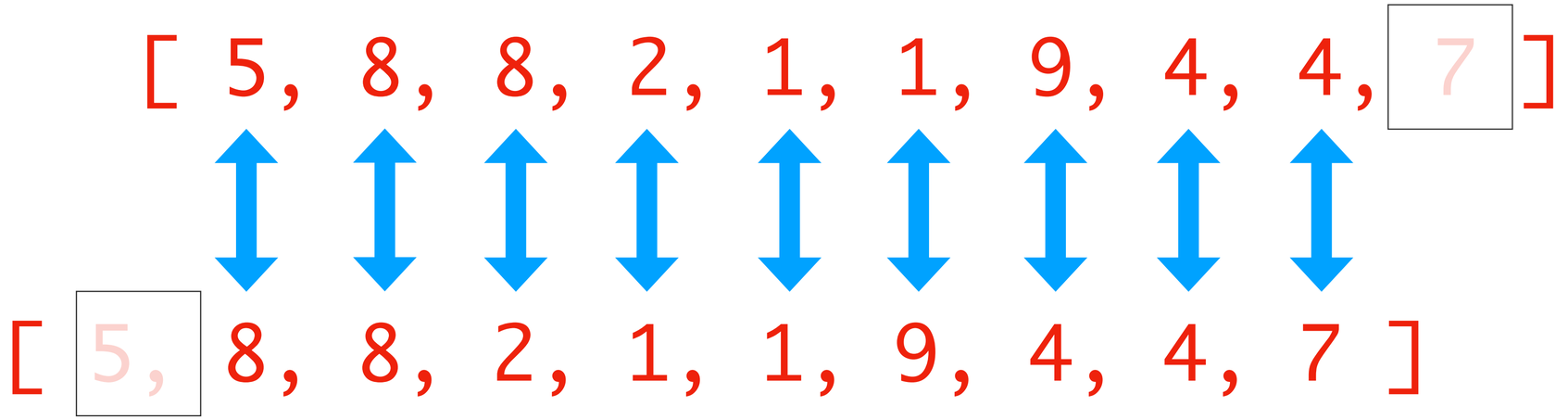
Visual hint:

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```

```
[ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]
```

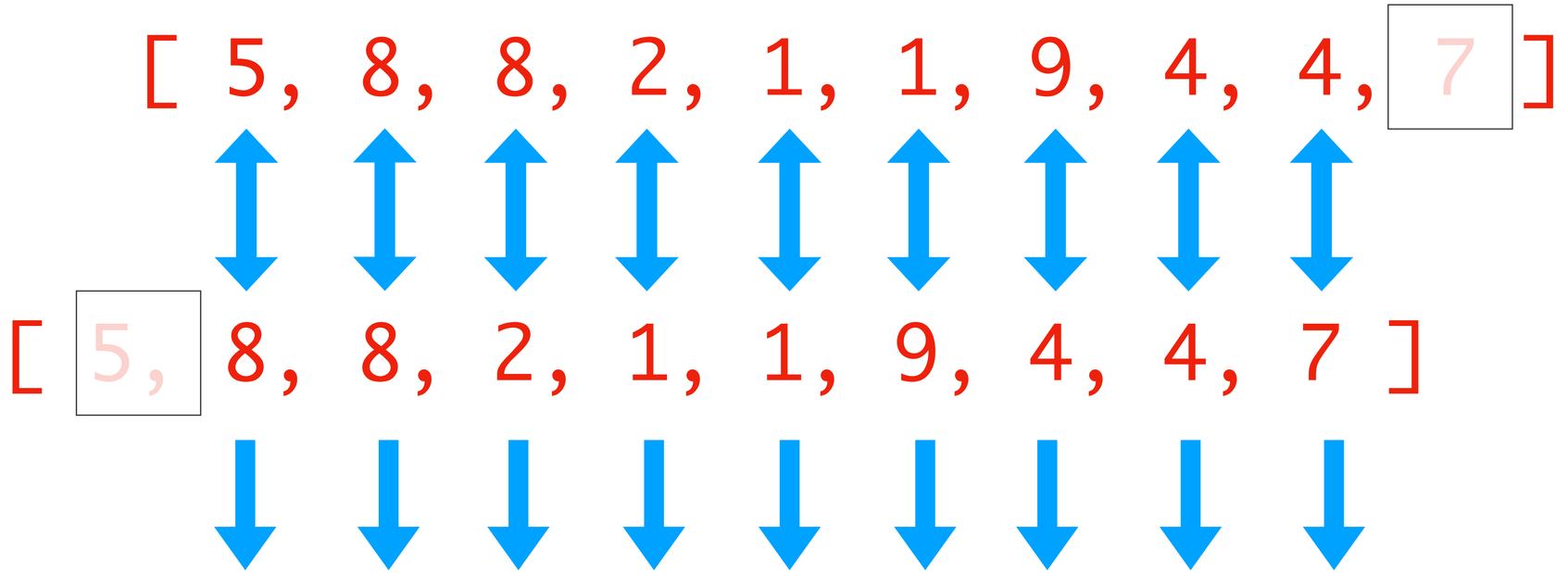
Counting adjacent repeated values in a sequence

Visual hint:



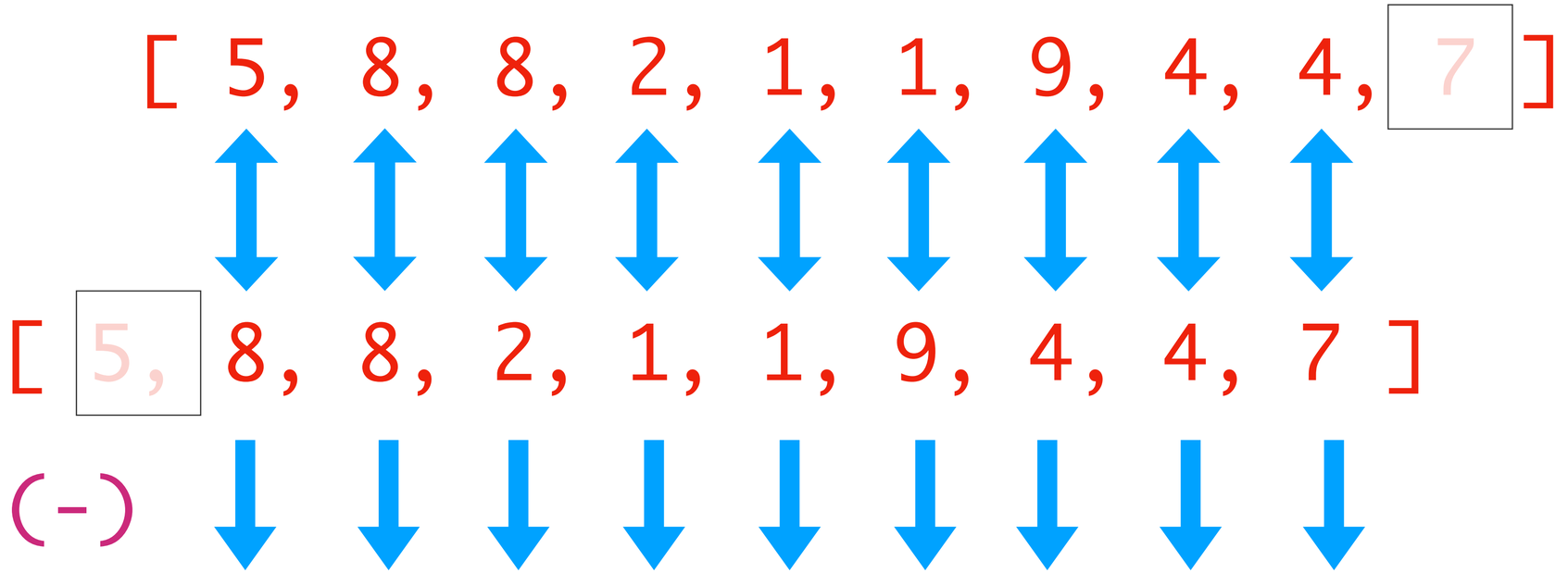
Counting adjacent repeated values in a sequence

Visual hint:



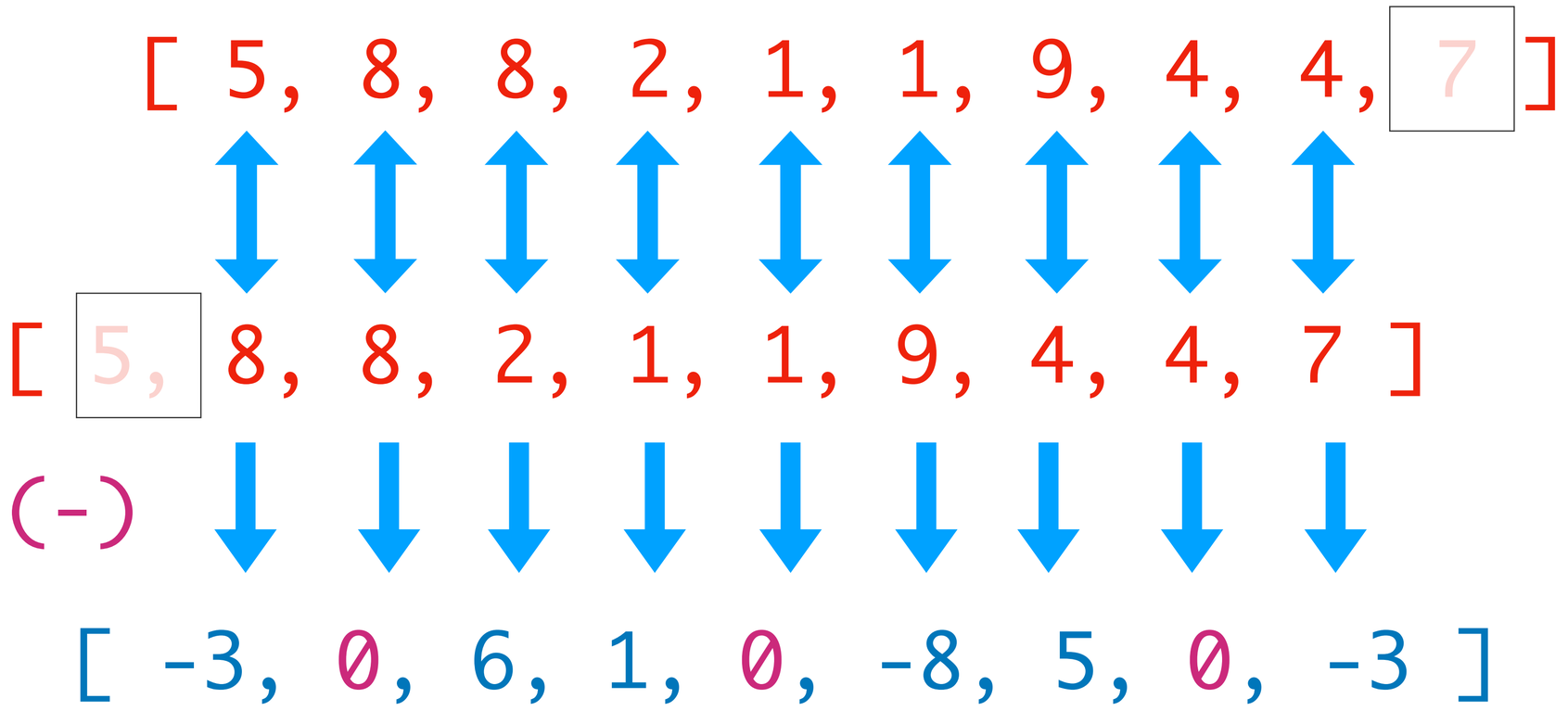
Counting adjacent repeated values in a sequence

Visual hint:



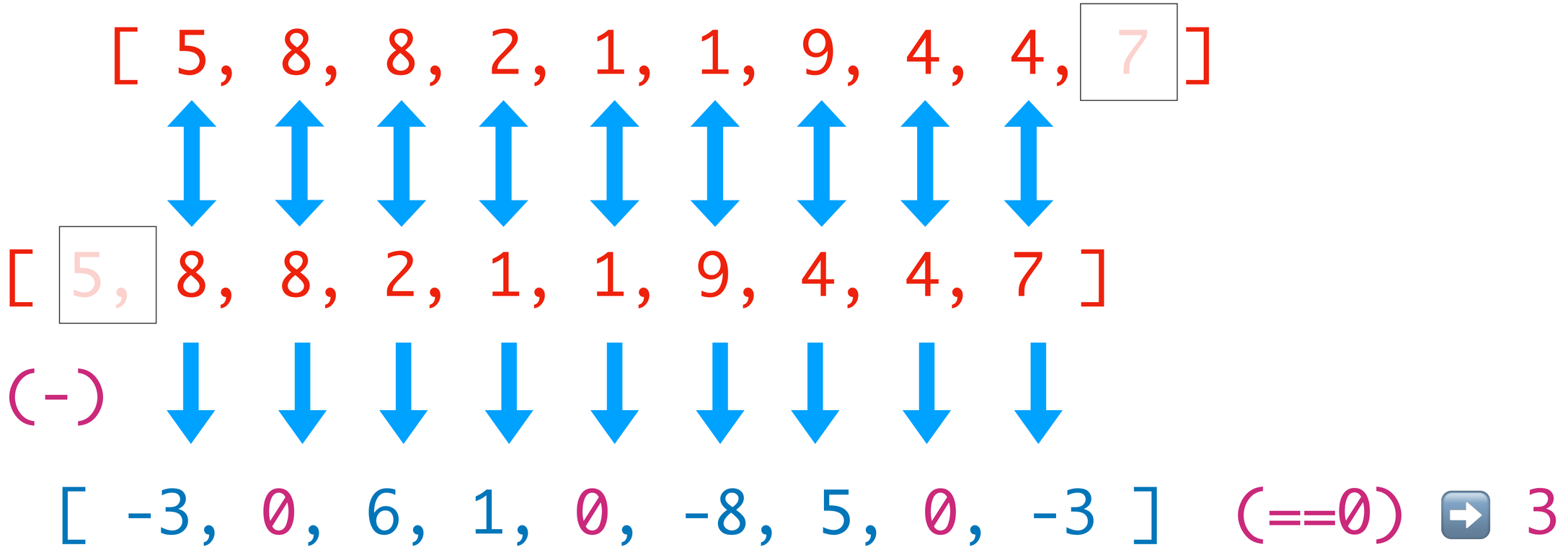
Counting adjacent repeated values in a sequence

Visual hint:



Counting adjacent repeated values in a sequence

Visual hint:



Counting adjacent repeated values in a sequence

```
let xs = [ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]

count_if f = length . filter f
adj_diff = mapAdjacent (-)
count_adj_equals = count_if (==0) . adj_diff

> count_adj_equals xs
3
```

That's it !

Counting adjacent repeated values in a sequence

Let's break it down:

```
// C++
[](auto a, auto b) { return a + b; }
plus{}

[](auto e) ->bool { return e == 1; }
```

```
// Haskell
(\a b -> a + b)
(+)

(\e -> e == 1)
(==1)
```

Lambdas & sections

Counting adjacent repeated values in a sequence

Let's break it down:

```
length::[a] -> Int
filter::(a->Bool) -> [a] -> [a]

=>

count_if::(a->Bool) -> [a] -> Int
count_if f = length . filter f
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
mapAdjacent::(a->a->b) -> [a] -> [b]
mapAdjacent _ [] = []
mapAdjacent f xs = zipWith f xs (tail xs)
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
mapAdjacent::(a->a->b) -> [a] -> [b]
mapAdjacent _ [] = []
mapAdjacent f xs = zipWith f xs (tail xs)
```

```
(-)::a -> a -> a
adj_diff = mapAdjacent (-)

=>

adj_diff::[a] -> [a]
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
(==0)::a -> Bool
count_if::(a->Bool) -> [a] -> Int
adj_diff::[a] -> [a]
```

```
count_adj_equals::[a] -> Int
count_adj_equals = count_if (==0) . adj_diff
```

Counting adjacent repeated values in a sequence

Let's break it down:

```
let xs = [ 5, 8, 8, 2, 1, 1, 9, 4, 4, 7 ]

> let ds = adj_diff xs
[ -3, 0, 6, 1, 0, -8, 5, 0, -3 ]

> count_if(==0) ds
3
```

Counting adjacent repeated values in a sequence

The algorithm

```
count_if f = length . filter f
adj_diff = mapAdjacent (-)
count_adj_equals = count_if (==0) . adj_diff
```

Back to modern C++

C++ Counting adjacent repeated values in a sequence

Back to modern C++

```
template <typename T>
int count_adj_equals(const T & xs)
{
  return accumulate(0,
           zip(xs, tail(xs)) | transform(equal_to{}));
}
```

C++ Counting adjacent repeated values in a sequence

Back to modern C++

```
template <typename T>
int count_adj_equals(const T & xs)
{
  return accumulate(0,
           zip(xs, tail(xs)) | transform(equal_to{}));
}
```

Ranges FTW