

+ 22

The Imperatives Must Go

VICTOR CIURA



The Imperatives Must Go!

CppCon

September 2022

 @ciura_victor

Victor Ciura
Senior SW Engineer
Visual C++



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`, value 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.

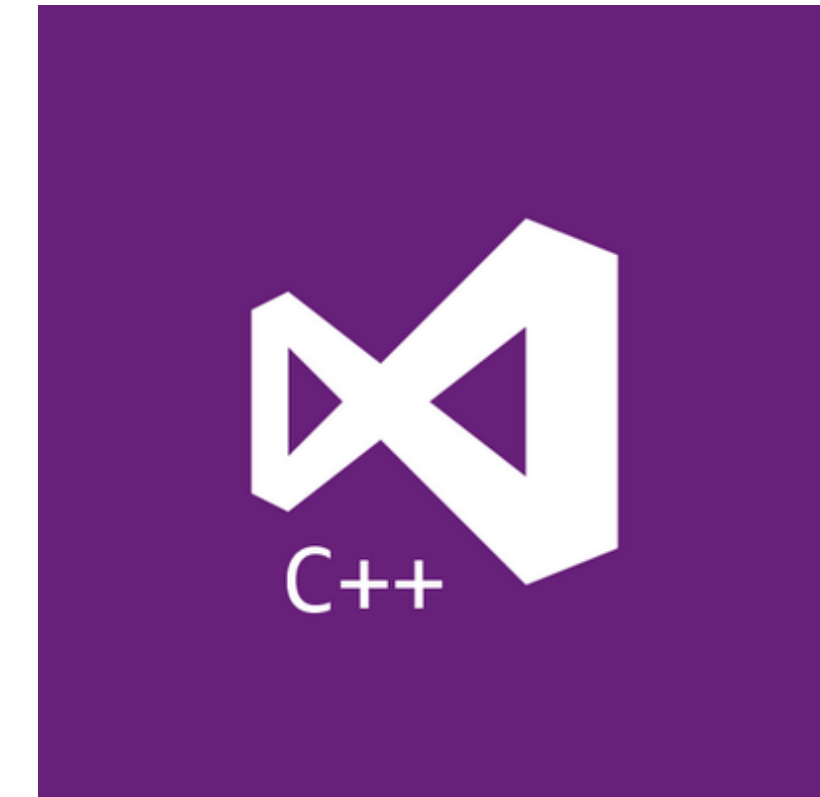
About me



Advanced Installer



Clang Power Tools



Visual C++

 [@ciura_victor](https://twitter.com/ciura_victor)

Welcome to CppCon 2022 !

Join #visual_studio channel on CppCon Discord
<https://aka.ms/cppcon/discord>

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey
<https://aka.ms/cppcon>



Do ask questions as we go along

Comments are welcome, too



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 😈





Functional Programming

What is it all about ?



pipelines

ranges

optional

IO monad

Maybe | Just

algorithms

lifting

monoids

lambdas & closures

fold

values types

lazy evaluation

declarative vs imperative

monads

algebraic data types

map

higher order functions

pattern matching

composition

FP

expressions vs statements

pure functions

category theory

currying

recursion

partial application

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**

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**


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


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

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

WHAT

Non-Functional

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

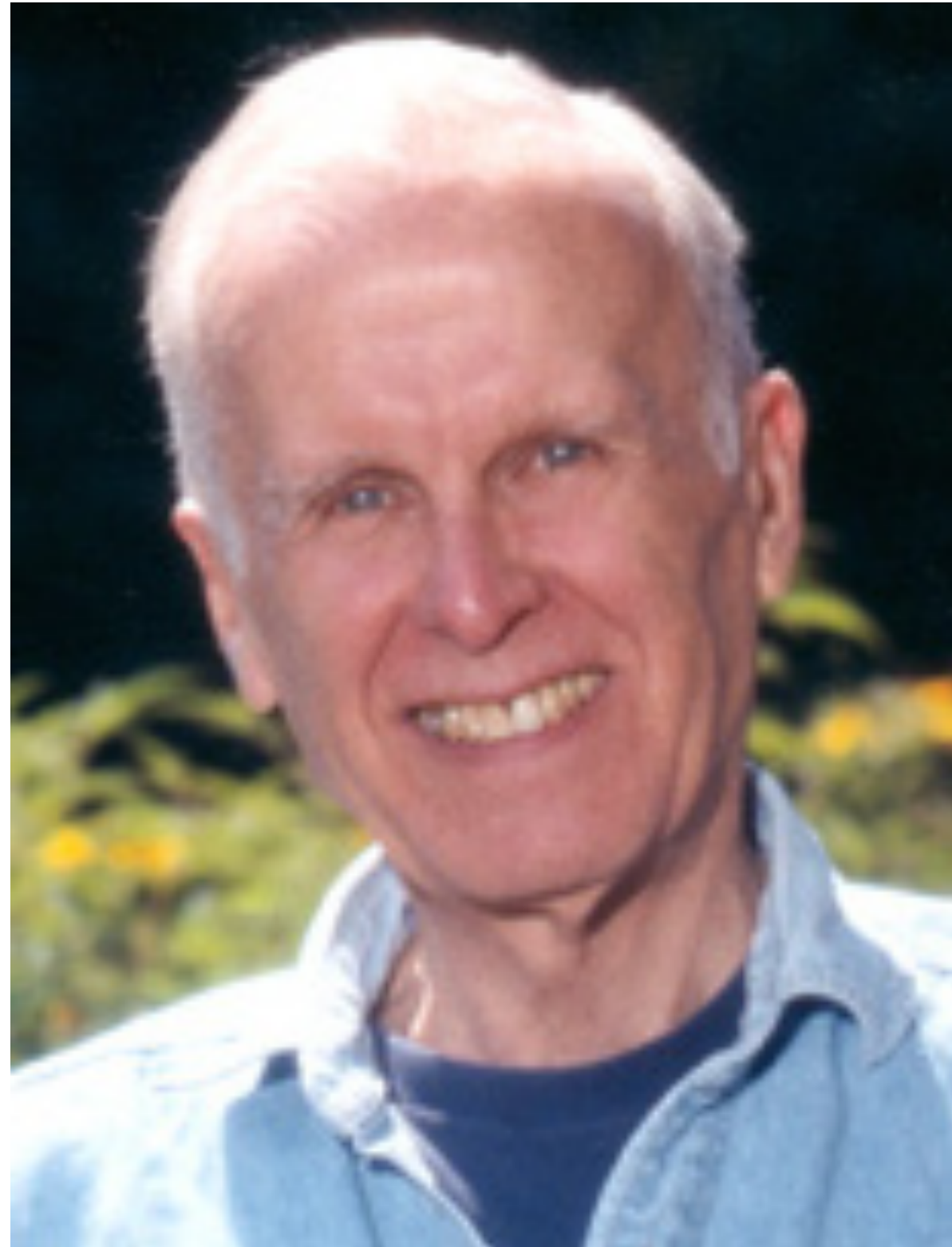
1960s



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

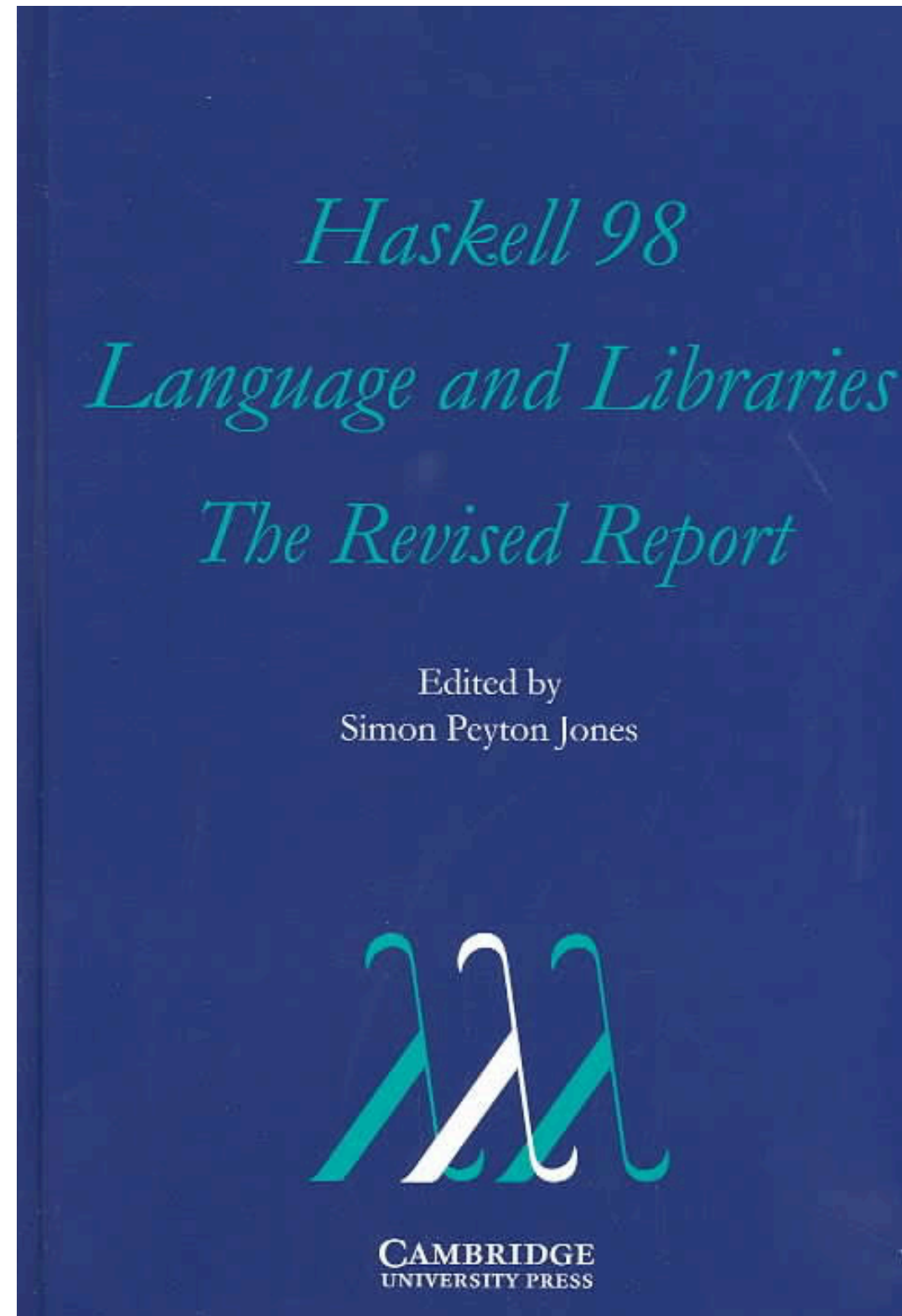


Phil Wadler and others develop **type classes** and **monads**,
two of the main innovations of Haskell

Historical Background

2003

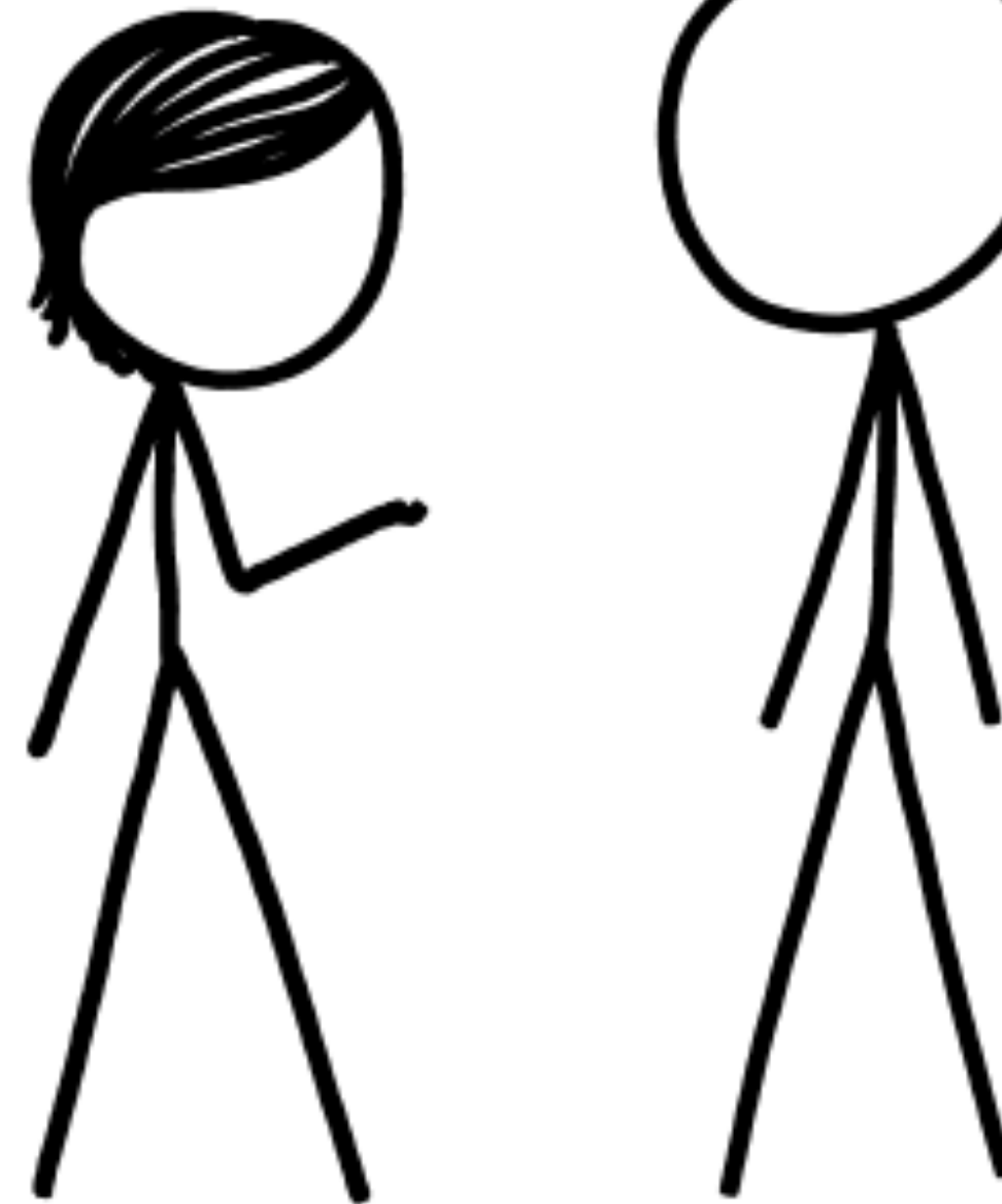
2010



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

CODE WRITTEN IN HASKELL
IS GUARANTEED TO HAVE
NO SIDE EFFECTS.

...BECAUSE NO ONE
WILL EVER RUN IT?



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

$$\begin{aligned} f [] &= [] \\ f (x:xs) &= f ys ++ [x] ++ f zs \\ &\text{where} \\ &\quad ys = [a \mid a \leftarrow xs, a \leq x] \\ &\quad zs = [b \mid b \leftarrow xs, b > x] \end{aligned}$$

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

q [3,2,4,1,5]



q [2,1] ++ [3] ++ q [4,5]



q [1] ++ [2] ++ q [] q [] ++ [4] ++ q [5]



[1]

[]

[]

[5]

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);

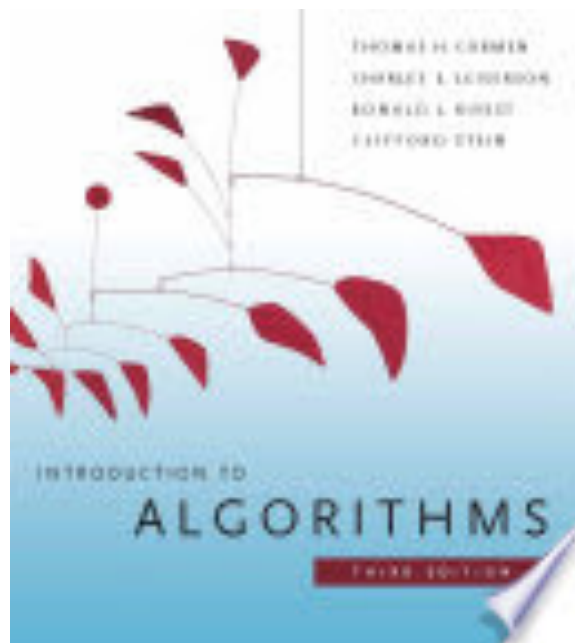
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
/* 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)
}
```



pseudo-code

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

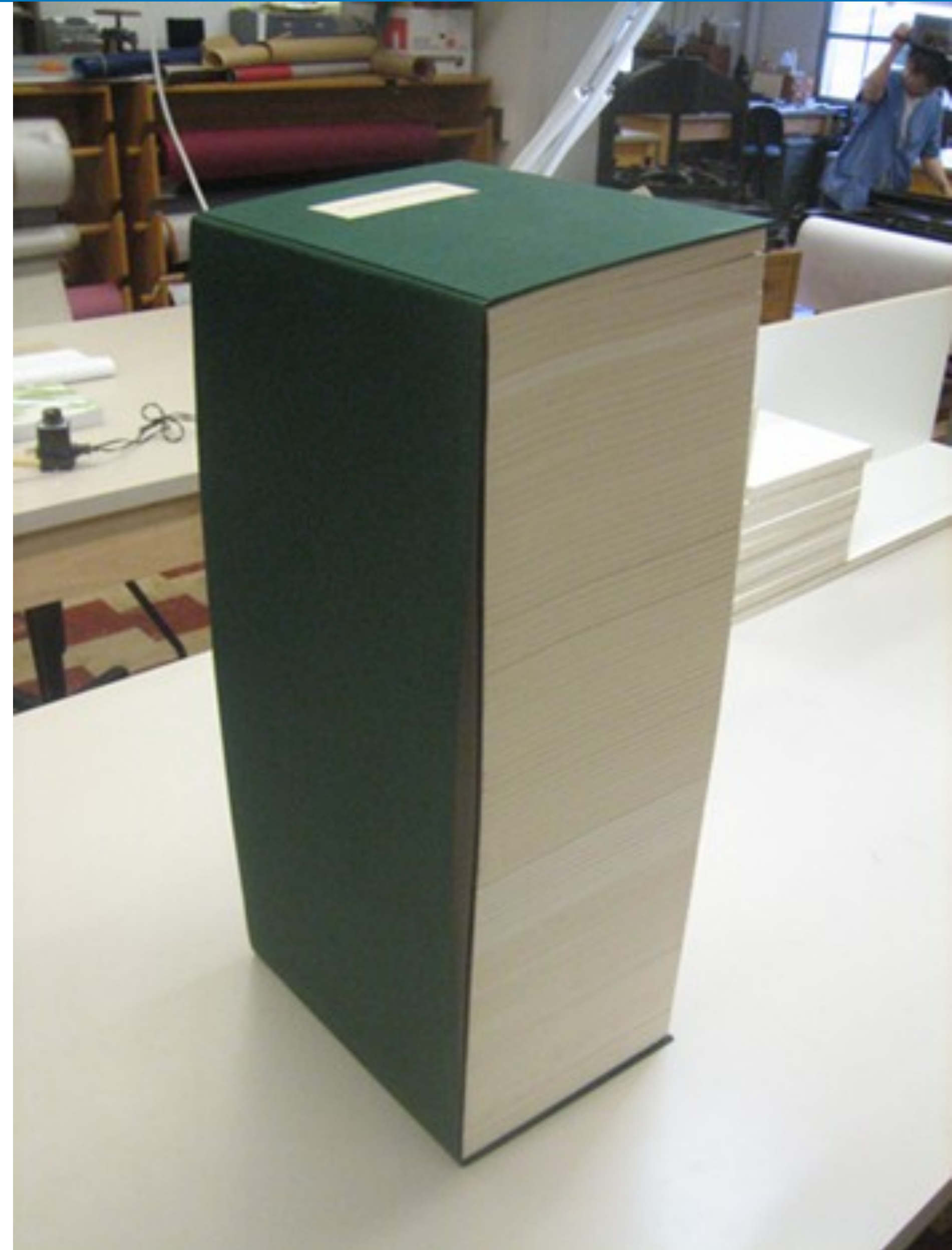
It's all about | pipelines

Taking inspiration from **Doug McIlroy**'s UNIX shell script,
write an algorithm in **your favorite programming language**,
that solves the same problem: **word frequencies**



How do I start on this journey?

Category Theory for Programmers



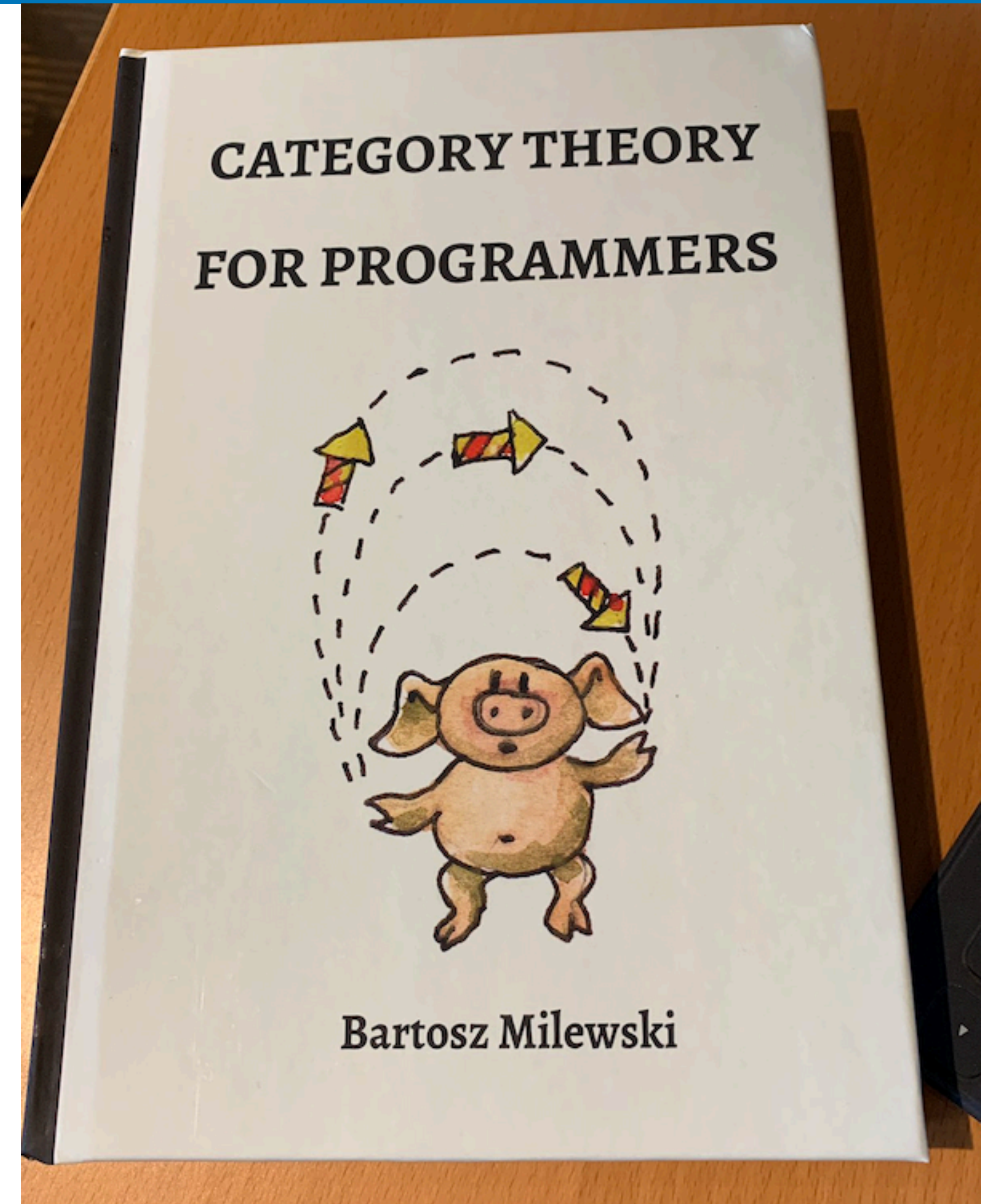
The Book

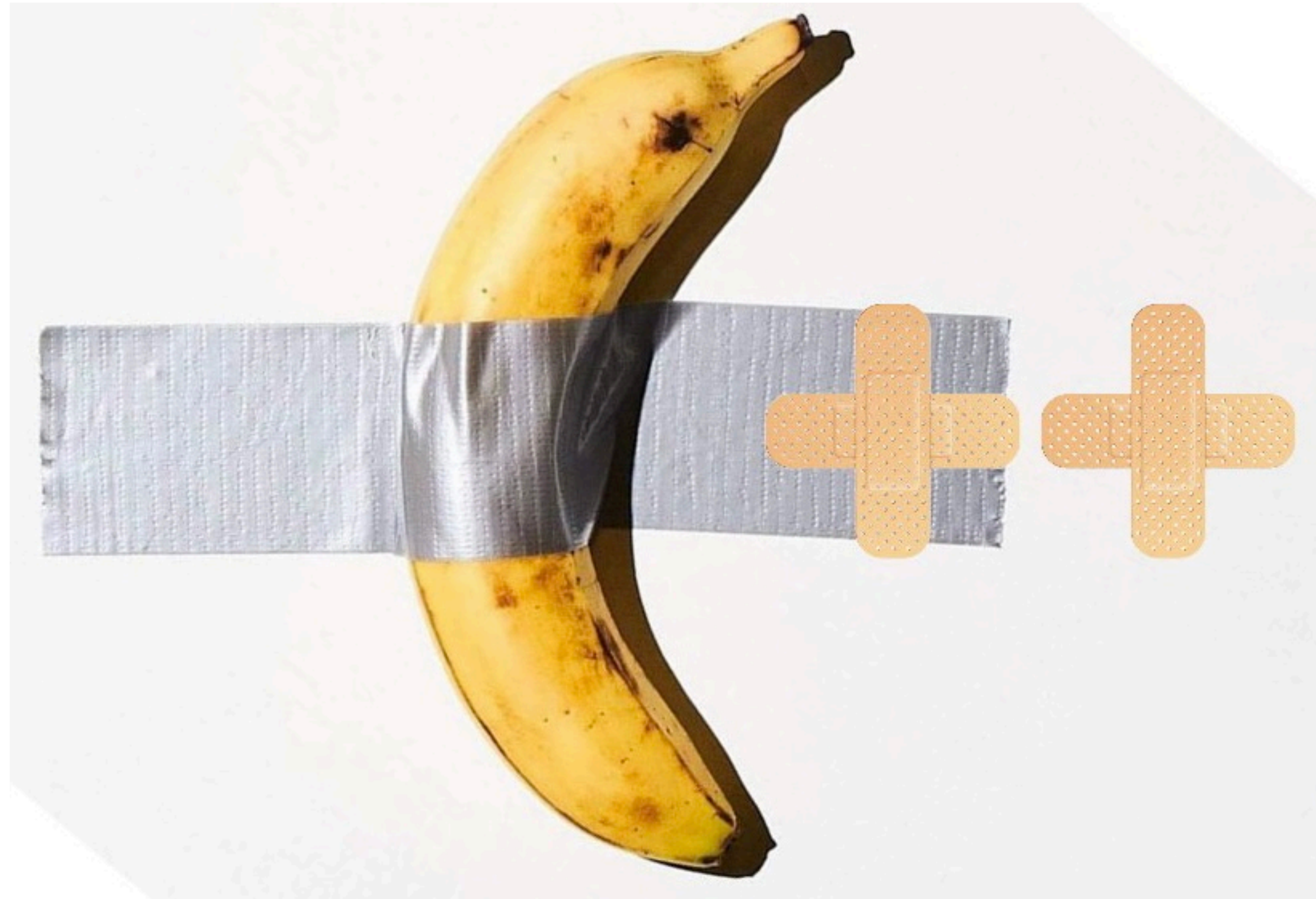


Bartosz Milewski

@BartoszMilewski

github.com/hmemcpy/milewski-ctfp-pdf





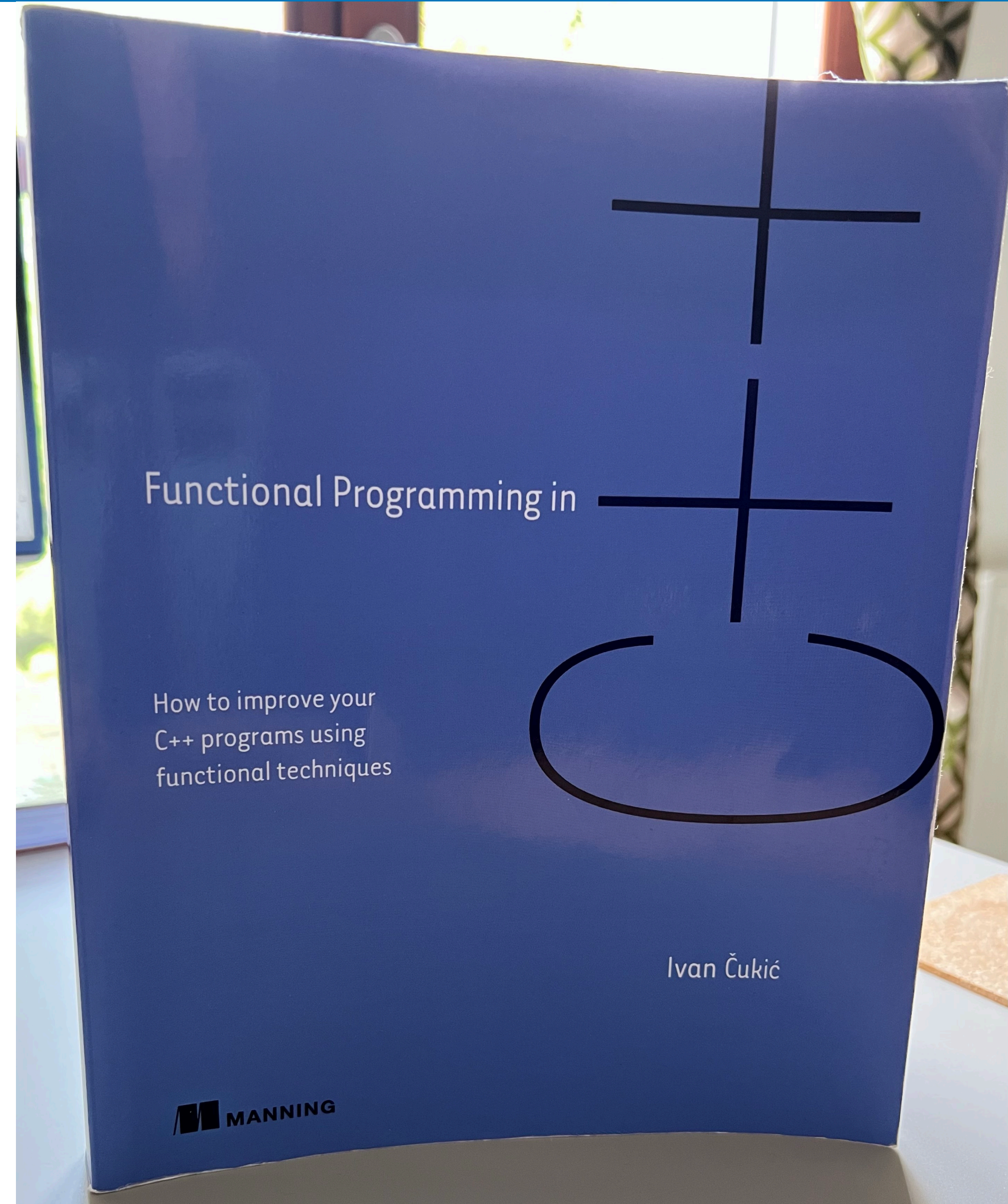
twitter.com/tvaneerd/status/1387

The Book



Ivan Čukić
@ivan_cukic

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



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 'function' 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)

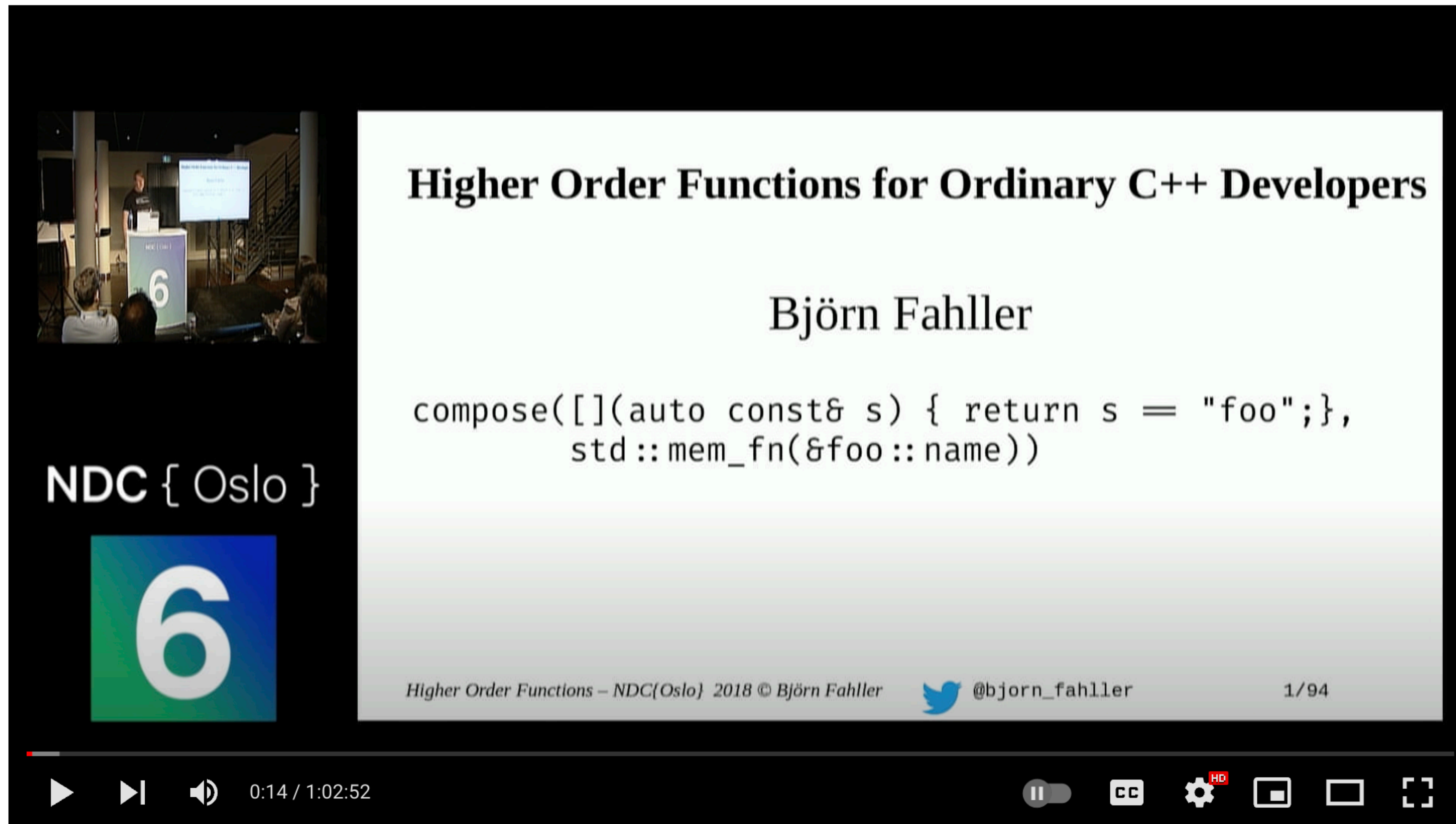
#define LIFT(lift_func) [](auto&& ... p) \
    LIFT_THRICE(lift_func(LIFT_FWD(p)...))
```

Need a lift?

Lifts overloaded functions named 'function' 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 C++ Developers

Björn Fahller

```
compose([](auto const& s) { return s == "foo"; },  
        std::mem_fn(&foo::name))
```

NDC { Oslo }

6

Higher Order Functions – NDC{Oslo} 2018 © Björn Fahller @bjorn_fahller 1/94

0:14 / 1:02:52

NDC Oslo 2018

Higher order functions for ordinary C++ developers - Björn Fahller

youtube.com/watch?v=kcBISmo3XIk



Boxes

Type Constructors

There are various ways to hide  a value:

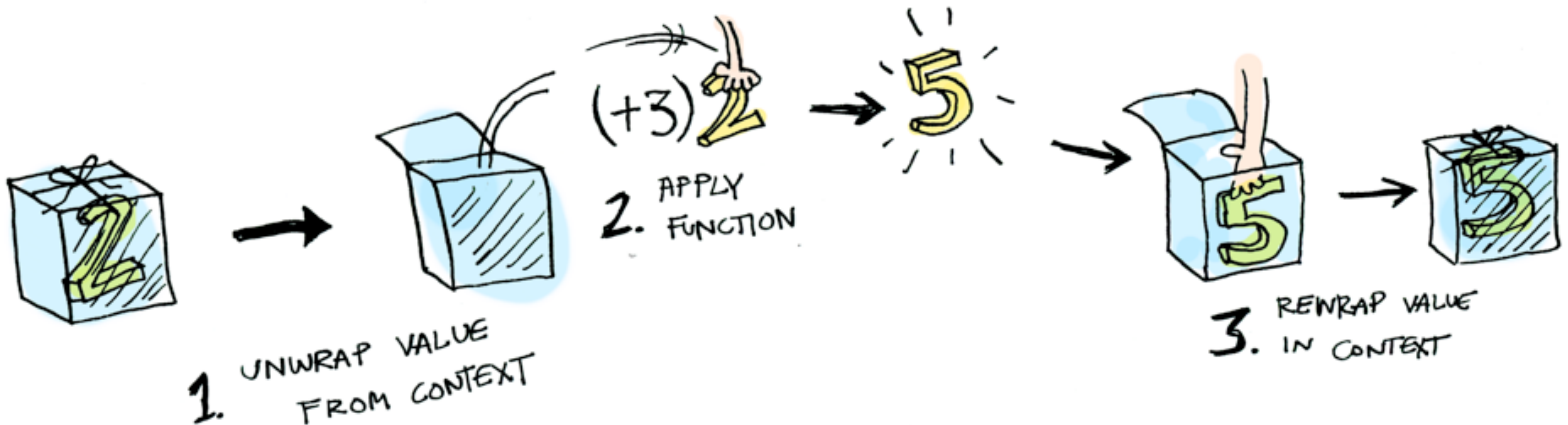
- `unique_ptr<T> p;`
- `shared_ptr<T> p;`
- `vector<T> v;`
- `optional<T> o;`
- `function<T(int)> f;`

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

`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

```
optional<T> f()
```

if / else

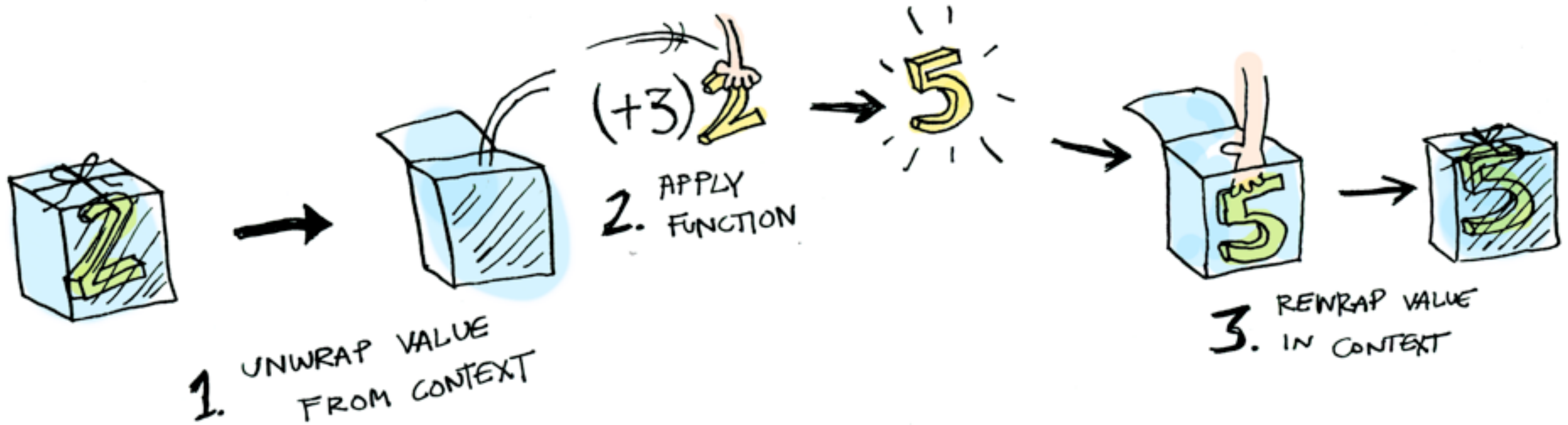
```
optional<T> g(optional<T> in)
```

if / else

```
optional<T> h(optional<T> in)
```

 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);
```

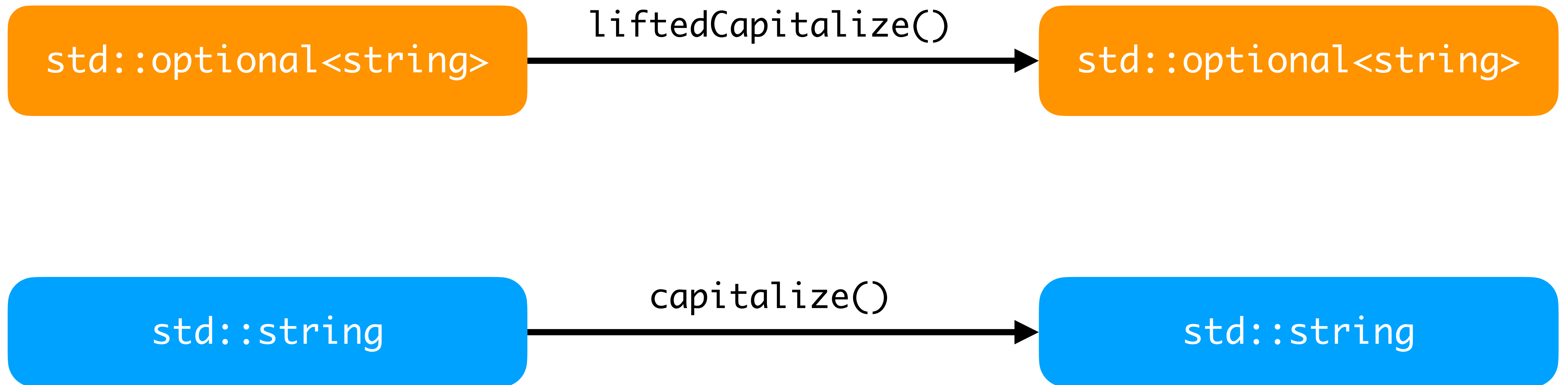
Lifting `capitalize()`

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()



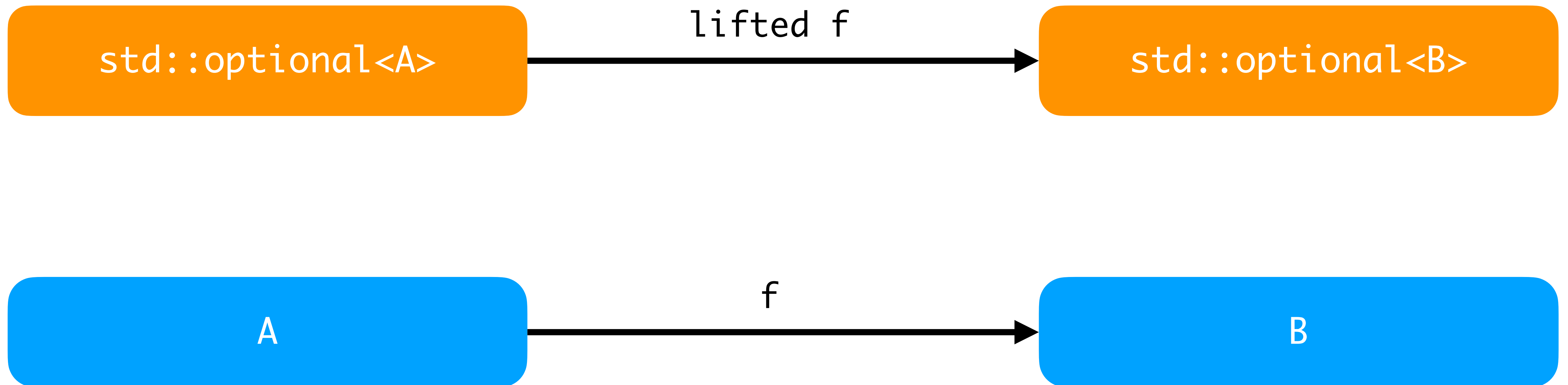
Lifting any function

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

```
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;
}
```

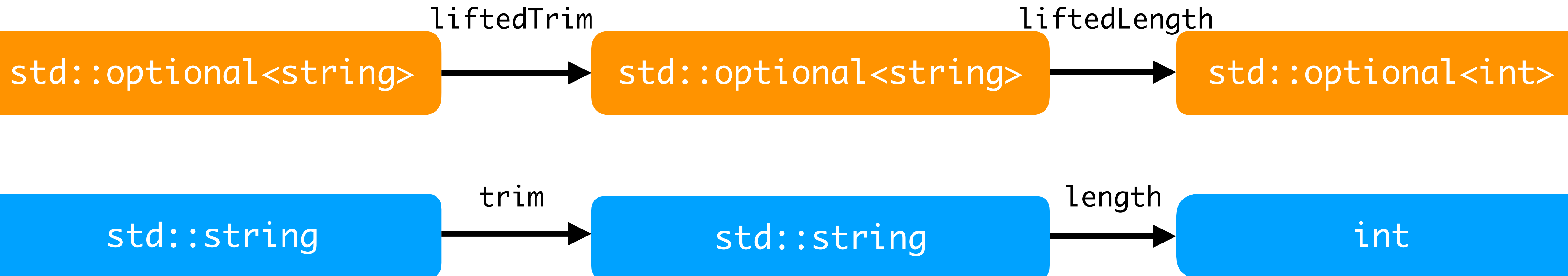

Lifting any function



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
```

```
...
```

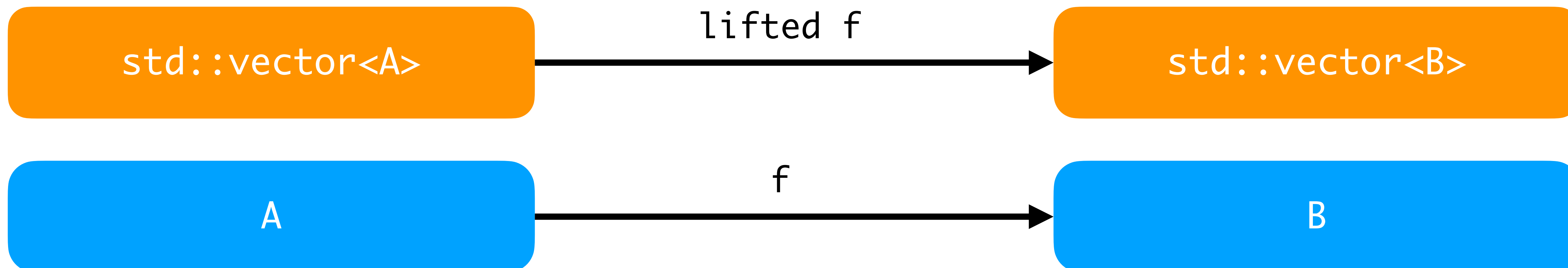
```
optional<string> debug_location()  
{  
    return current_pc  
        | fmap(dsym::load_symbol)  
        | fmap(dsym::to_string);  
}
```

Lifting a function to a vector

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

```
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 $A \rightarrow B$ create a function `box<A> -> box`

Why?

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

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;
}
```

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()
```


C++ now 2018 MAY 7-11 cppnow.org



Ben Deane

Easy to Use,
Hard to Misuse
Declarative Style in C++

REPLACING CONDITIONALS

Style	Signature Element	Elimination Strategy
Imperative	Statement	multi-computation
Object-Oriented	Object construction	polymorphism
Functional	Function call	higher order function
Generic	Type instantiation	traits class

The Conditional-Replacement Meta-Pattern.

115 / 122

Expressions yield values, Statements do not;

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/

The image shows a video player interface. The main content area displays a dark slide with the text "value semantics" in large white font. In the top right corner of the video frame, the text "cppcon | 2018" is visible, with "THE C++ CONFERENCE • BELLEVUE, WASHINGTON" below it. To the right of the main slide, there is a smaller video inset showing a man, Juan Pedro Bolívar Puente, speaking. Below the inset, his name "JUAN PEDRO BOLÍVAR PUENTE" is written in white. Further down, the title of the talk "The Most Valuable Values" is displayed. At the bottom of the video player, there is a control bar with play, next, volume, and progress (1:21 / 58:51) icons, as well as a settings gear icon and a "HD" indicator.

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

[youtube.com/watch?v= oBx_NbLghY](https://youtube.com/watch?v=oBx_NbLghY)

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.



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

youtube.com/watch?v=SAMR5GJ_GqA

Immutable DS

The slide features the title "RADIX BALANCED SEARCH" in pink and white text. Below it, the binary string "v[17] → 01 00 01" is displayed. A diagram shows a binary tree with three levels of nodes, each represented as a 4-bit vector. The root node is [0, 1, 0, 0]. The left child is [0, 1, 0, 0], and the right child is [0, 1, 1, 0]. The left child's left child is [0, 1, 0, 0], and its right child is [0, 1, 1, 0]. The right child's left child is [0, 1, 1, 0], and its right child is [0, 1, 1, 0]. The leaf nodes are labeled with letters: a b c d, e f g h, i j k l, m n o p, q r s t, u v y. A play button is overlaid on the right child node. The speaker, Juan Pedro Bolivar Puente, is shown in a video inset on the right. The video player interface at the bottom shows a progress bar at 0:00 / 1:05:59 and various control icons.

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

RADIX BALANCED SEARCH

v[17] → 01 00 01

JUAN PEDRO BOLIVAR PUENTE

Postmodern immutable data structures

0:00 / 1:05:59

CppCon 2017

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

youtube.com/watch?v=sPhpelUfu8Q

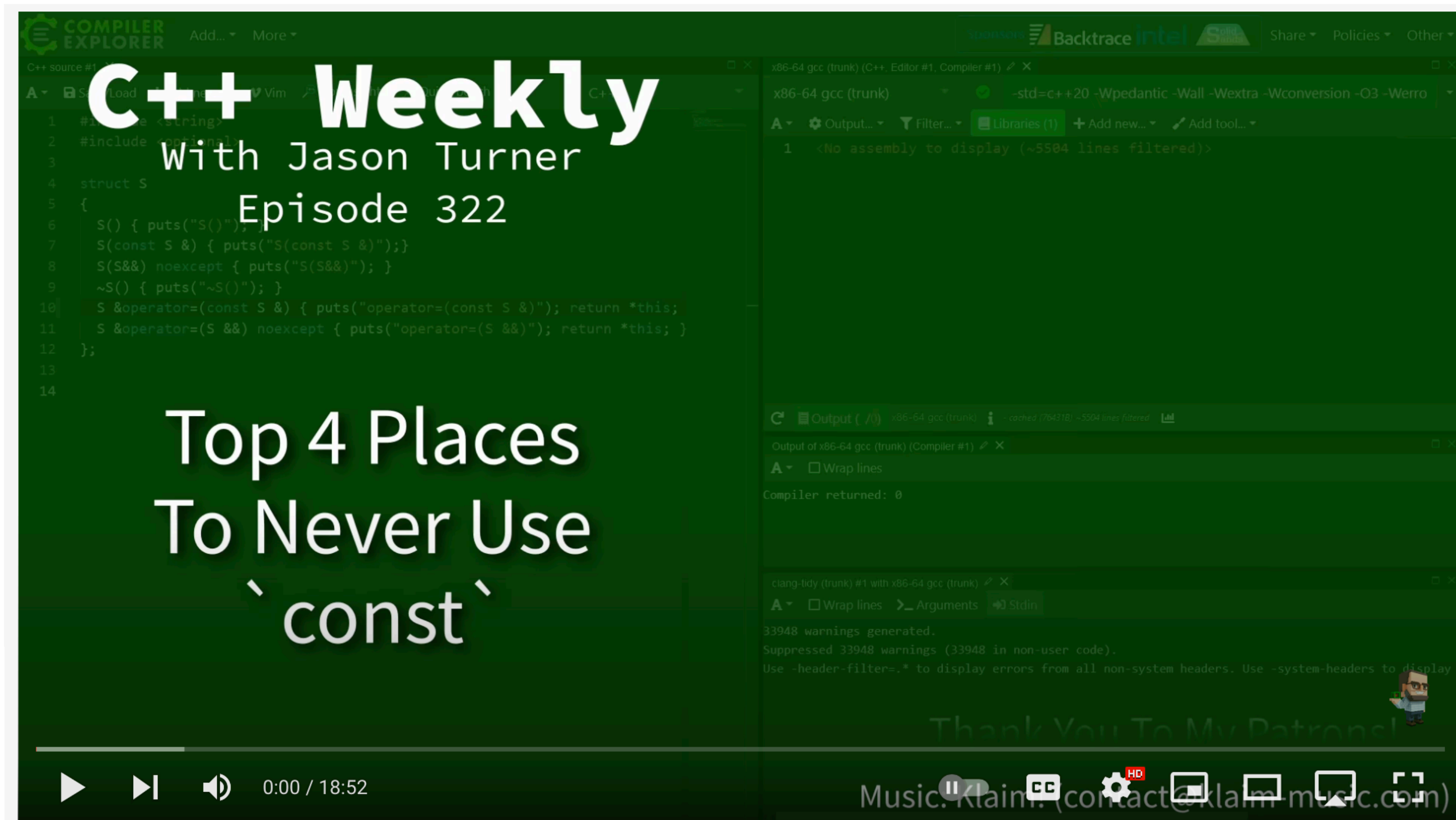
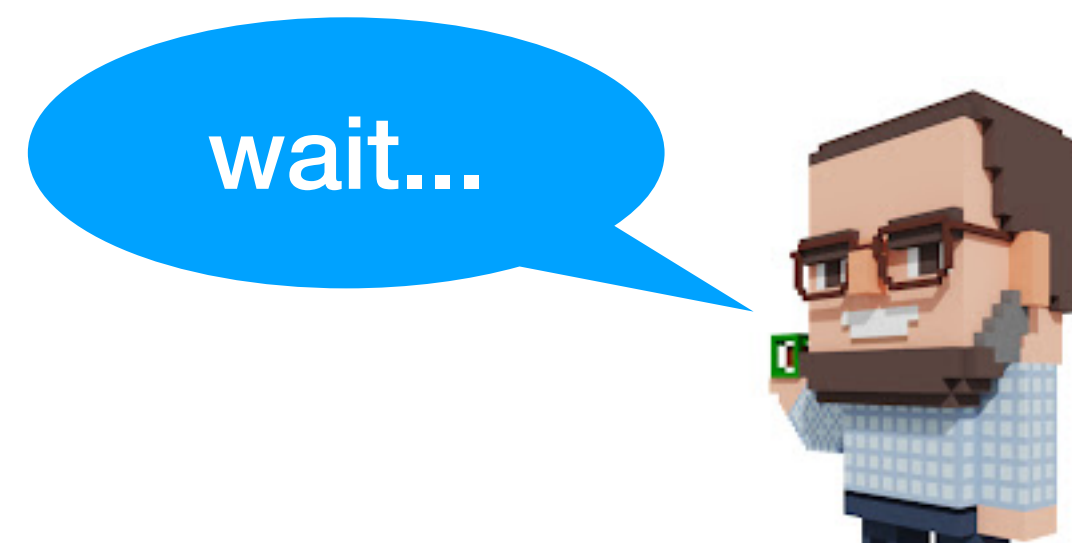
Adding **const** always helps

const all the things!



Adding **const** always helps

<https://www.youtube.com/watch?v=dGCxMmGvocE>



C++ Weekly - Ep 322 - Top 4 Places To Never Use `const`

Immutable



compiler-explorer.com/z/9Wcc54r9x



Top 4 places to **never use const**:

compiler-explorer.com/z/9Wcc54r9x



Top 4 places to **never use const**:

compiler-explorer.com/z/9Wcc54r9x



Top 4 places to **never use const**:

- don't `const` non-reference return types



Top 4 places to **never use const**:

- don't `const` **non-reference return types**
- don't `const` **local values** that need take advantage of implicit **move-on-return** operations (even if you have multiple different objects that might be returned)



Top 4 places to **never use const**:

- don't `const` **non-reference return types**
- don't `const` **local values** that need take advantage of implicit **move-on-return** operations (even if you have multiple different objects that might be returned)
- don't `const` non-trivial **value parameters** that you might need to **return directly** from the function



Top 4 places to **never use const**:

- don't `const` **non-reference return types**
- don't `const` **local values** that need take advantage of implicit **move-on-return** operations (even if you have multiple different objects that might be returned)
- don't `const` non-trivial **value parameters** that you might need to **return directly** from the function
- don't `const` **any member data**
 - it breaks implicit and explicit moves
 - it breaks common use cases of assignment

compiler-explorer.com/z/9Wcc54r9x

C++ 20 Ranges

The beginning of the end for [begin, end)

Jeff Garland

New algorithms

Many adaptors

Pipelines

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

```
for (auto const i : v
     | rv::reverse
     | rv::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 = std::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
     | rv::drop(2)
     | rv::take(3)
     | rv::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(std::begin(v), std::end(v));  
  
v.erase(  
    std::unique(std::begin(v), std::end(v)),  
    std::end(v));  
  
std::reverse(std::begin(v), std::end(v));
```

```
vector<int> v{ 21, 1, 3, 8, 13,  
             1, 5, 2 };  
  
v = std::move(v) |  
    ra::sort |  
    ra::unique |  
    ra::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<std::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 = rs::iota_view(101, 201)
        | rv::reverse
        | rv::filter([](auto v) { return v%7==0; })
        | rv::transform(to_string)
        | rv::take(3)
        | rs::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
```

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


Word frequencies

```
const auto results = words
| view::group_by(std::equal_to())
| view::transform([] (const auto & group) {
    const auto b = std::begin(group);
    const auto e = std::end(group);
    const auto size = std::distance(b, e);
    const std::string word = *b;
    return make_pair(size, word);
})
| ranges::to_vector | action::sort;
```

Word frequencies

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

Remember him?

1990s



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

Takeaway



<*> 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

Enjoy the rest of the conference!

Join #visual_studio channel on CppCon Discord
<https://aka.ms/cppcon/discord>

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey
<https://aka.ms/cppcon>

Our sessions

Monday 12th

- GitHub Features Every C++ Developer Should Know – Michael Price
- The Imperatives Must Go – Victor Ciura
- What's New in C++ 23 – Sy Brand
- C++ Dependencies Don't Have to Be Painful – Augustin Popa
- How Microsoft Uses C++ to Deliver Office – Zachary Henkel

Tuesday 13th

- High-performance Load-time Implementation Selection – Joe Bialek, Pranav Kant
- C++ MythBusters – Victor Ciura

Wednesday 14th

-memory-safe C++ - Jim Radigan

Thursday 15th

- What's New for You in Visual Studio Code – Marian Luparu, Sinem Akinci
- Overcoming Embedded Development Tooling Challenges – Marc Goodner
- Reproducible Developer Environments – Michael Price

Friday 16th

- What's New in Visual Studio 2022 – Marian Luparu, Sy Brand
- C++ Complexity (Keynote) – Herb Sutter

The Imperatives Must Go!

CppCon

September 2022

 @ciura_victor

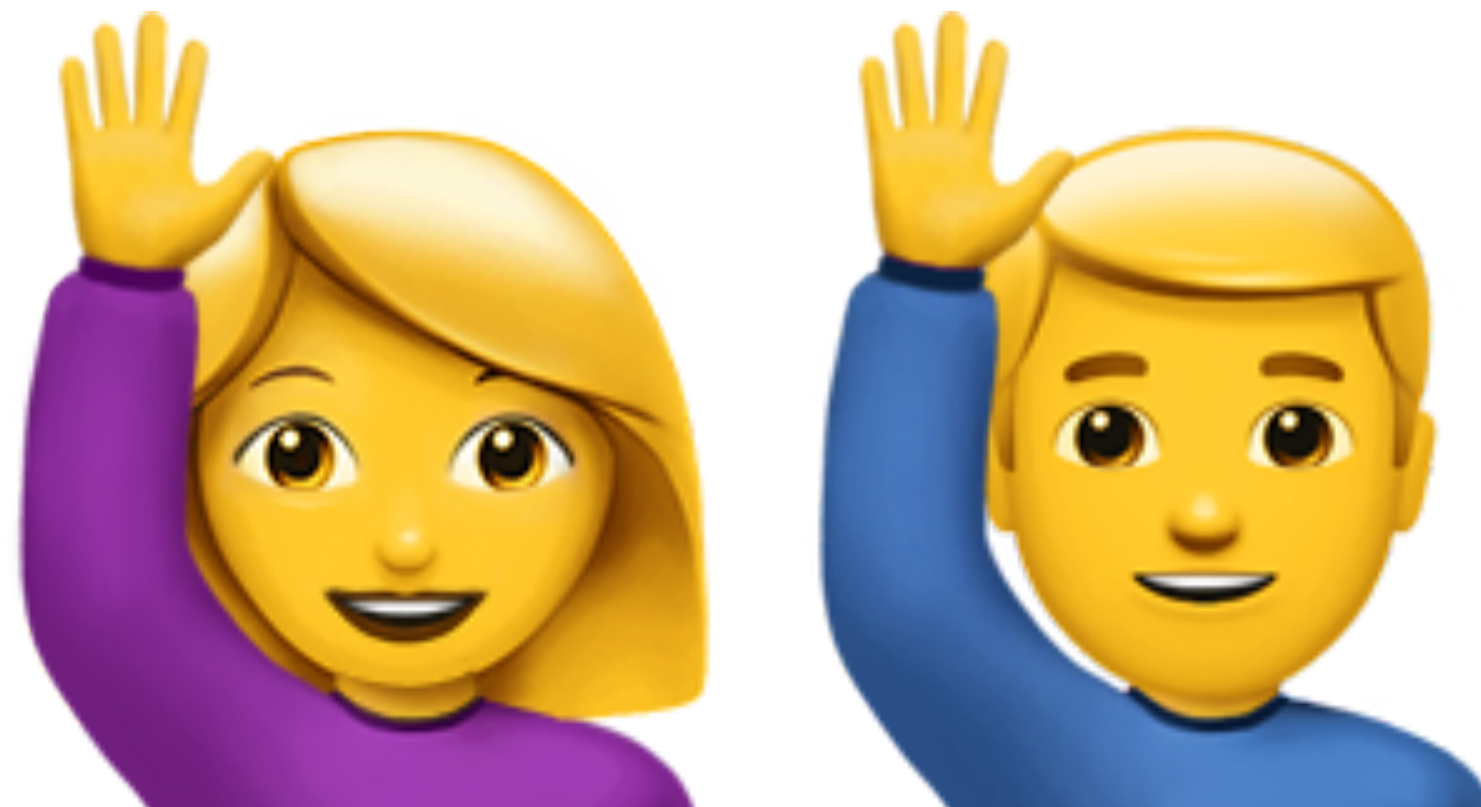
Victor Ciura
Senior SW Engineer
Visual C++



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

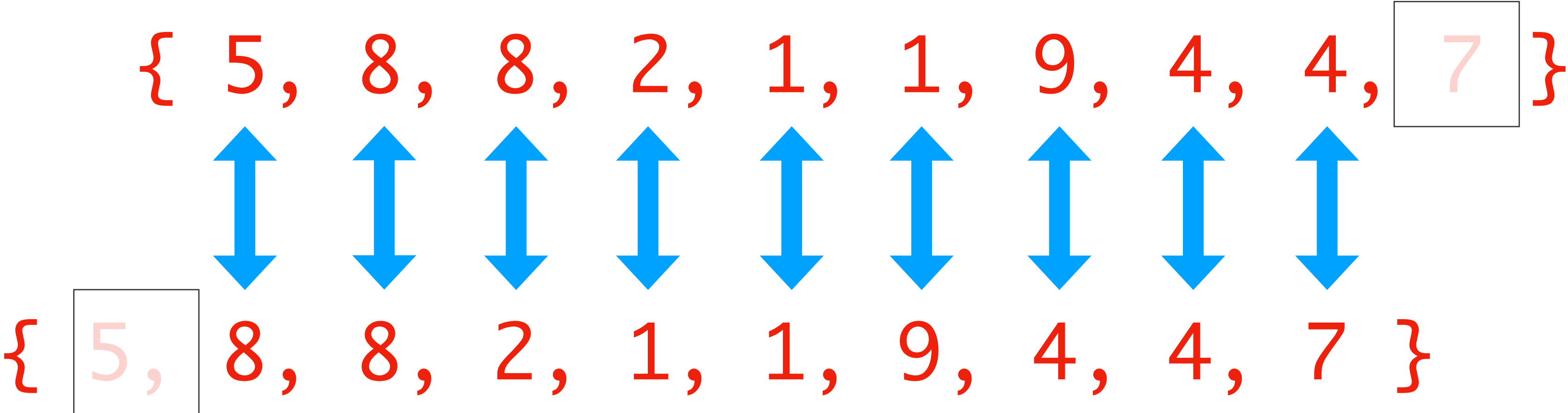
Visual hint:

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

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

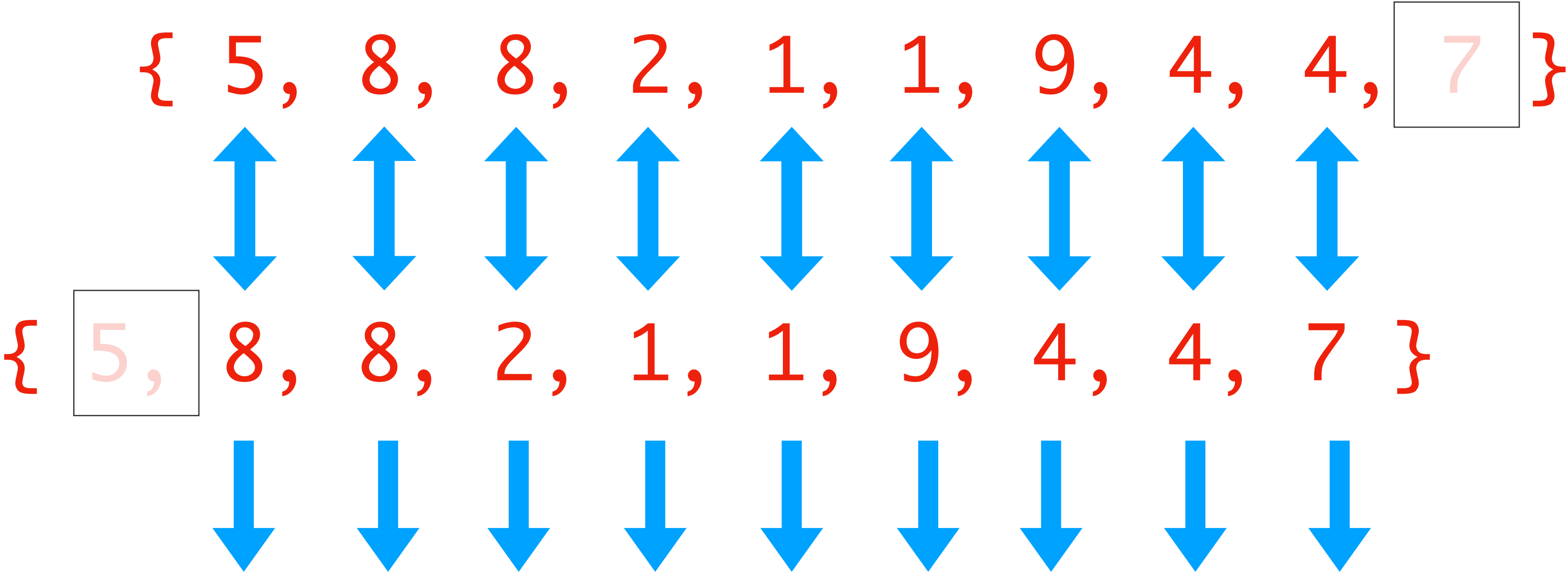
Counting adjacent repeated values in a sequence

Visual hint:



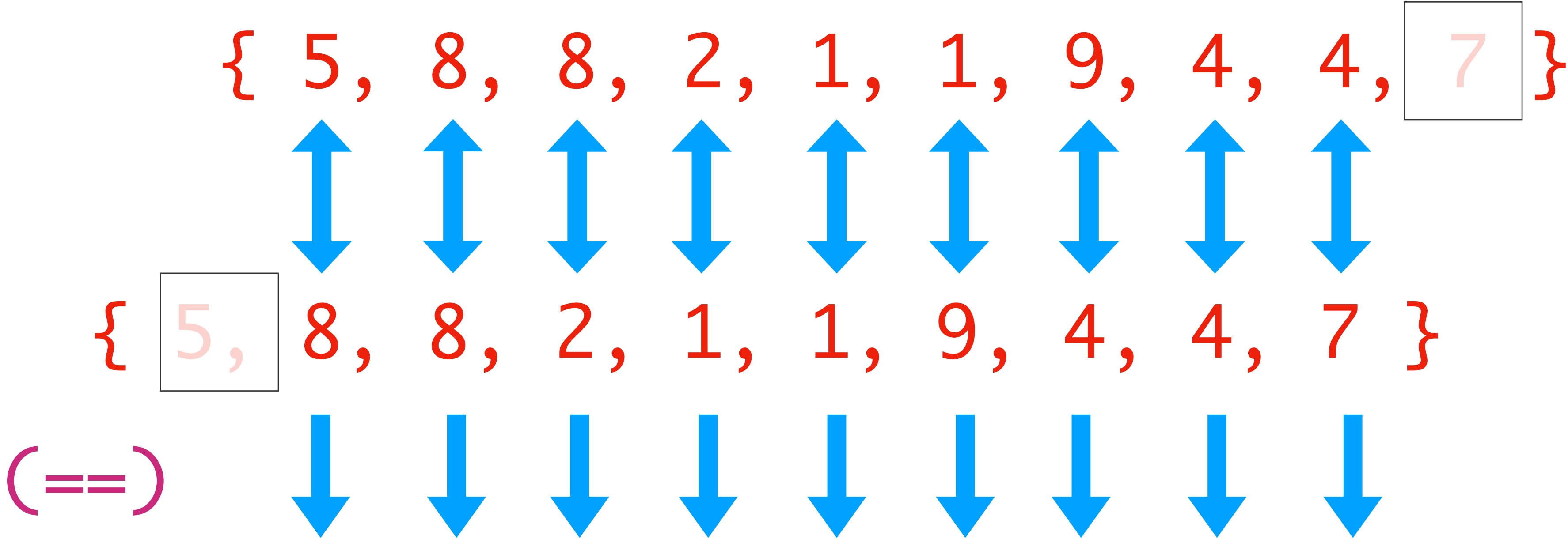
Counting adjacent repeated values in a sequence

Visual hint:



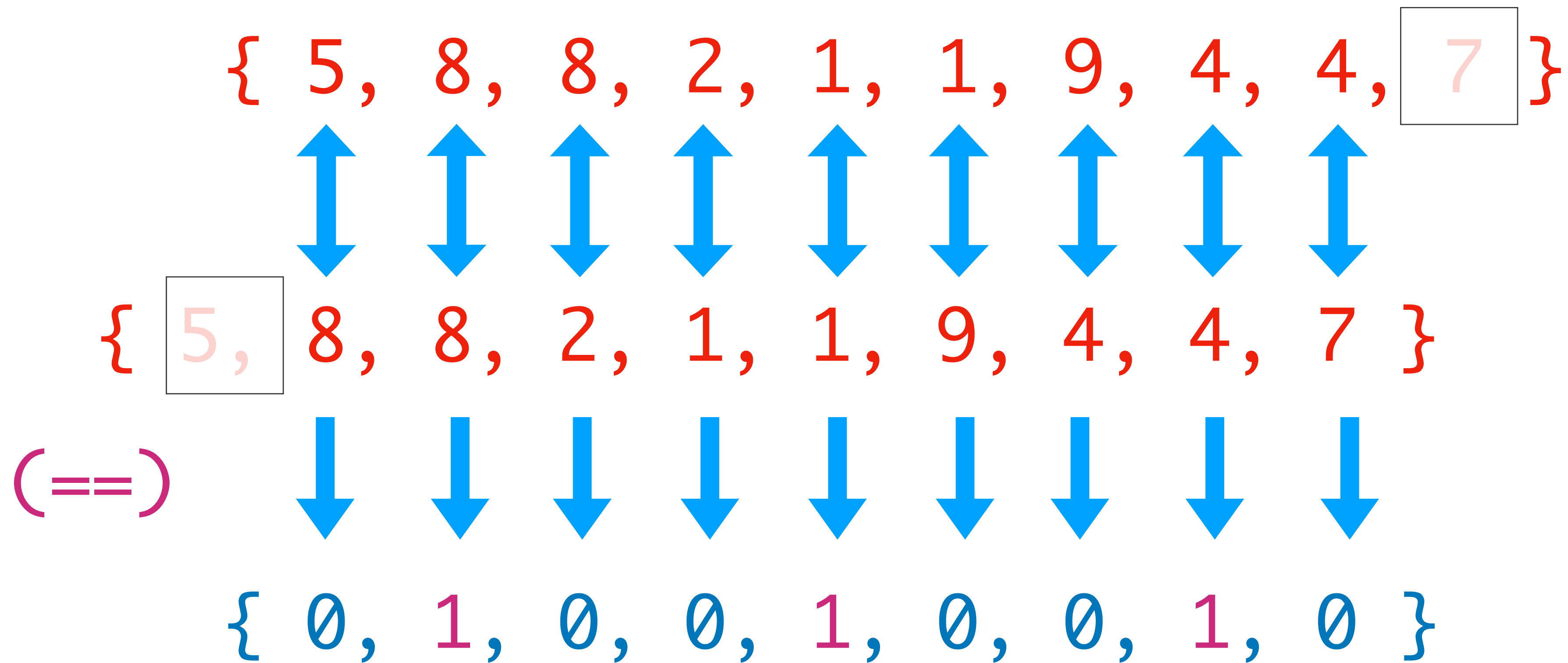
Counting adjacent repeated values in a sequence

Visual hint:



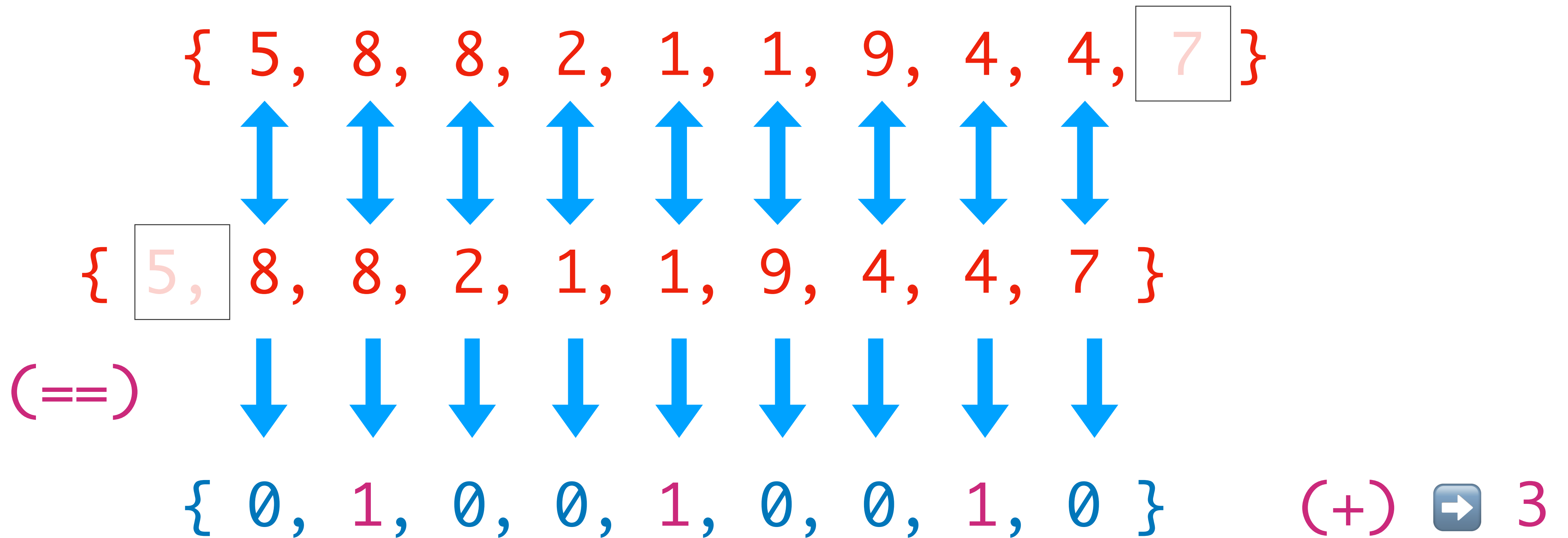
Counting adjacent repeated values in a sequence

Visual hint:



Counting adjacent repeated values in a sequence

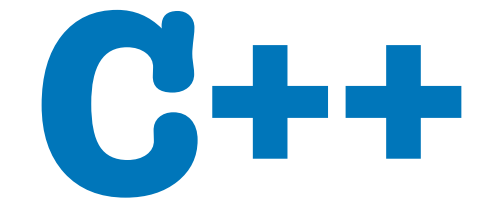
Visual hint:



C++

Counting adjacent repeated values in a sequence

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

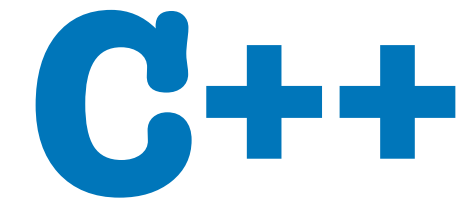


Counting adjacent repeated values in a sequence

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

How about something shorter ?

An STL `algorithm` maybe ?



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;  
}
```

C++ Counting adjacent repeated values in a sequence

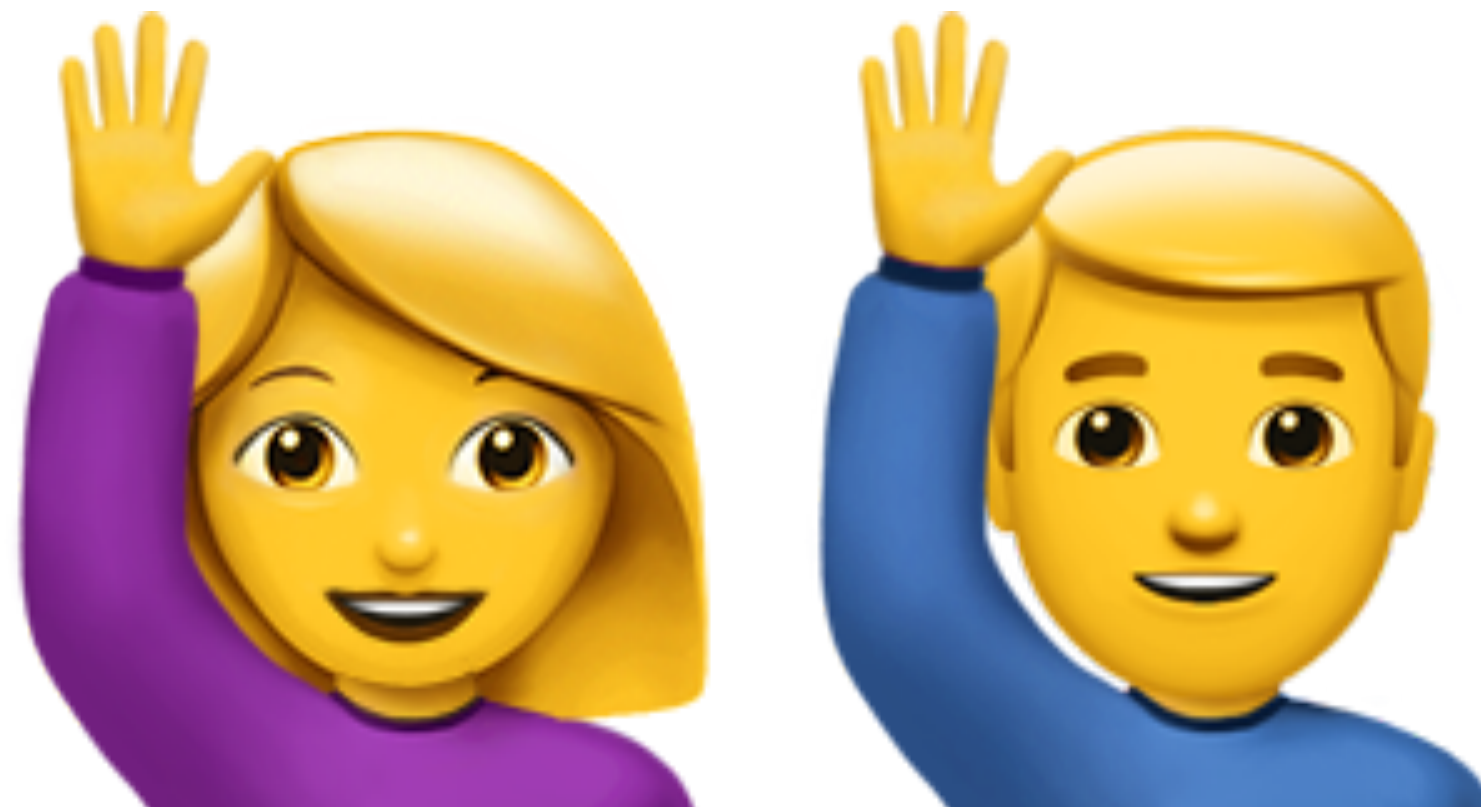
```
template <typename T>
int count_adj_equals(const T & xs) // requires non-empty range
{
    return std::inner_product(
        std::cbegin(xs), std::cend(xs) - 1, // to penultimate elem
        std::cbegin(xs) + 1,                // collection tail
        0,
        std::plus{},
        std::equal_to{});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

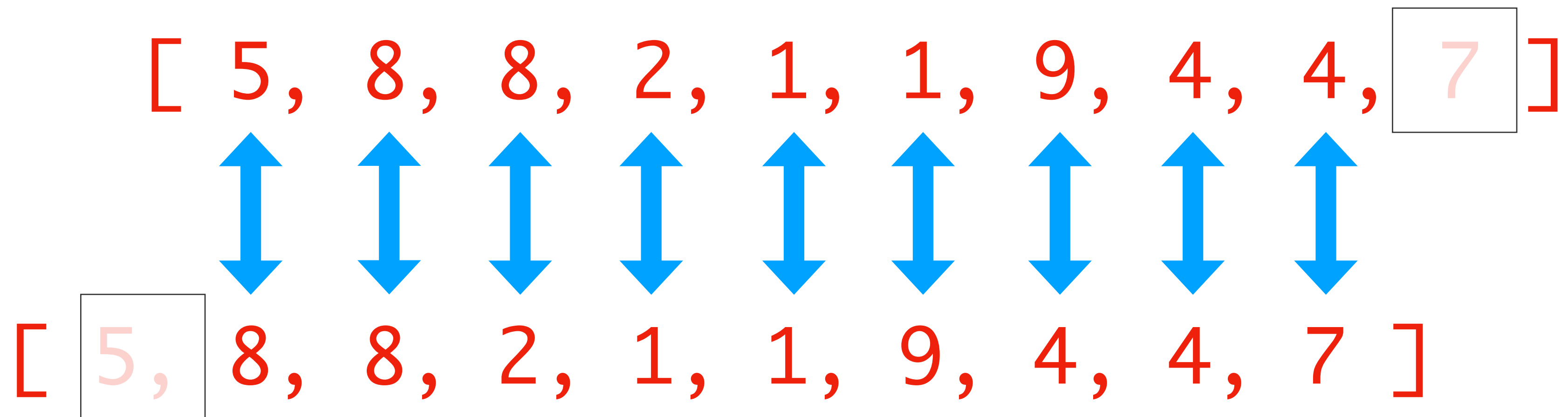
[5, 8, 8, 2, 1, 1, 9, 4, 4, 7]

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



Counting adjacent repeated values in a sequence

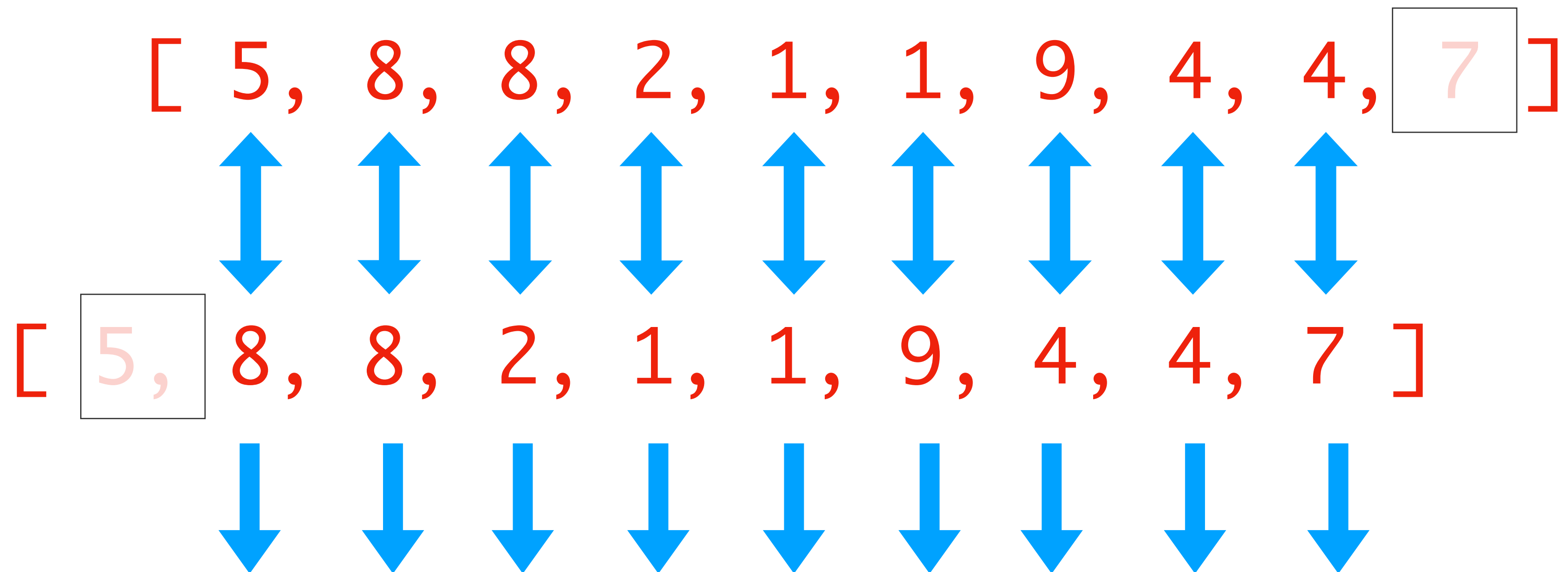
Visual hint:





Counting adjacent repeated values in a sequence

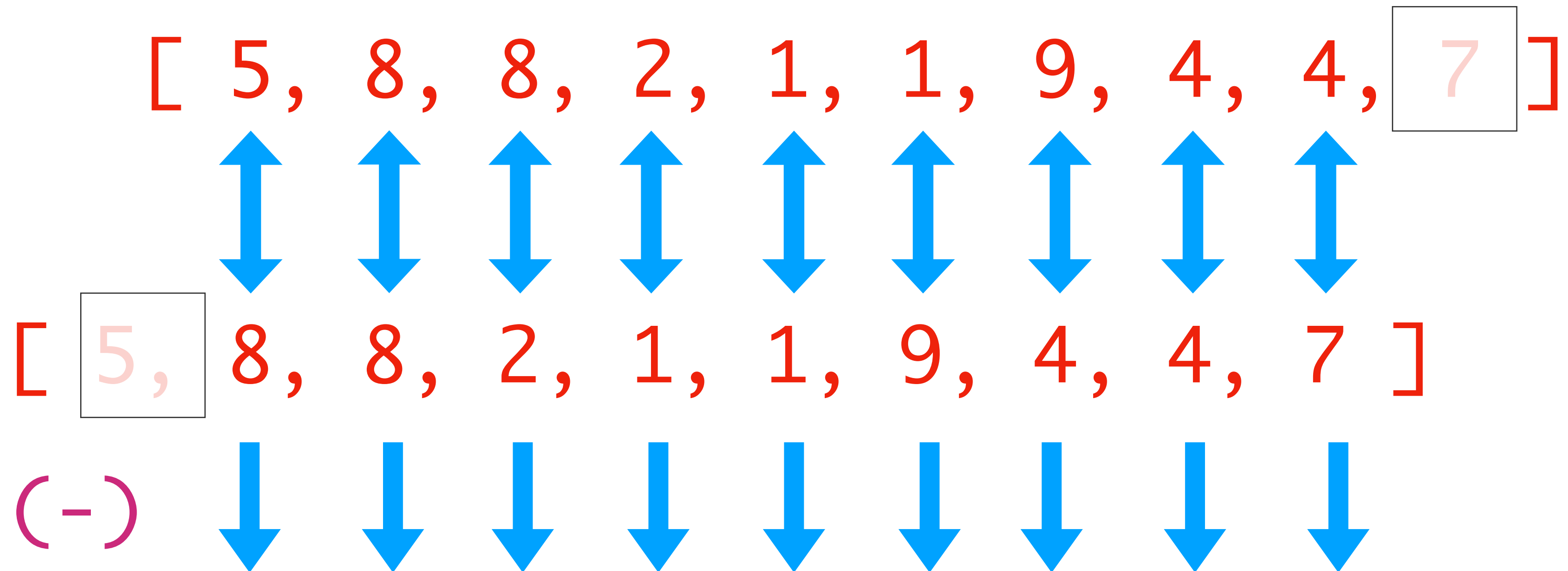
Visual hint:





Counting adjacent repeated values in a sequence

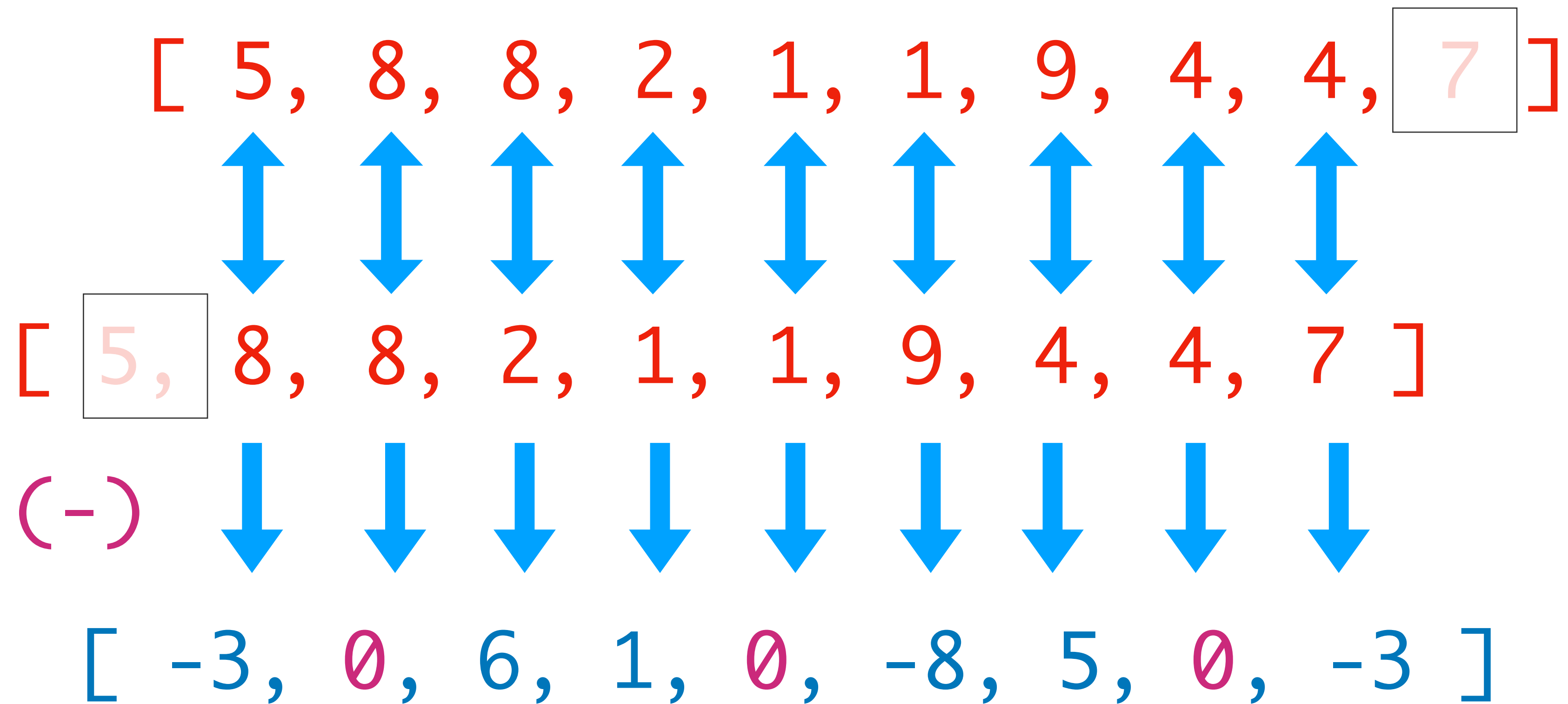
Visual hint:





Counting adjacent repeated values in a sequence

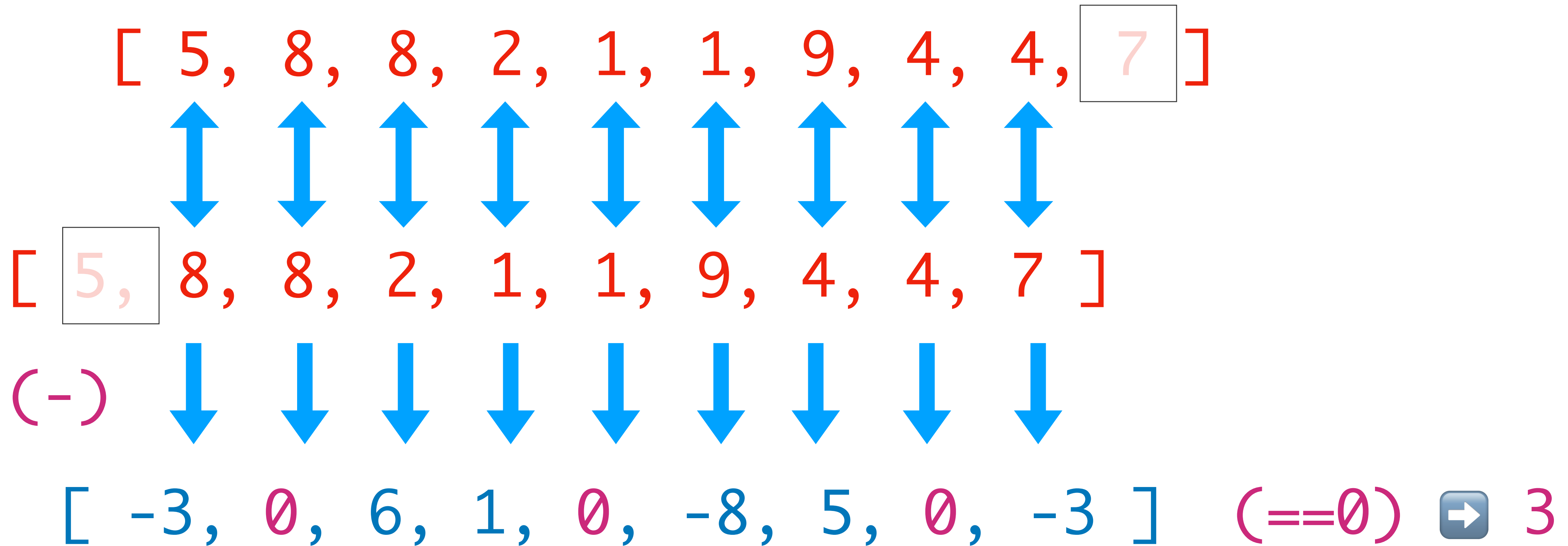
Visual hint:





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
```

C++

Counting adjacent repeated values in a sequence

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