

The Imperatives Must Go!



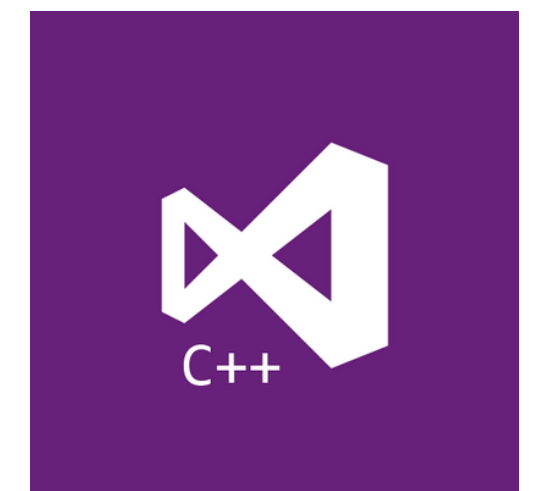
Core C++

June 2023

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

 [@ciura_victor@hachyderm.io](mailto:ciura_victor@hachyderm.io)

Victor Ciura
Principal 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.

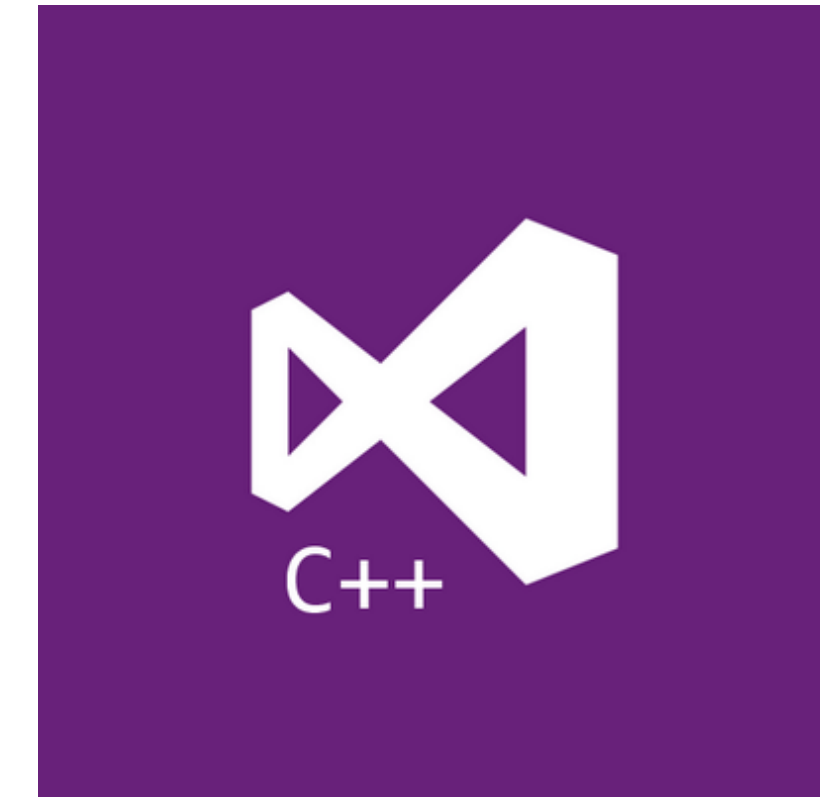
About me



Advanced Installer



Clang Power Tools



Visual C++

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

Functional Programming ... in C++ 🤔

What is it all about ?


```
void ImProcFunctions::Lab_Local(
int call, int sp, float** shbuffer, LabImage * original, LabImage * transformed, LabImage * reserved, LabImage * savenormtm, LabImage * savenormreti, LabImage * lastorig, int fw, int fh, int cx, int cy, int ow, int oh, int sk,
const LocretigainCurve& locRETgainCcurve, const LocretitransCurve& locRETtransCcurve,
const LUTf& lllocalcurve, bool locallutili,
const LUTf& cllocalcurve, bool localclutili,
const LUTf& lclocalcurve, bool localcutili,
const LocLHCurve& loclhCurve, const LocHHCurve& lochhCurve, const LocCHCurve& locchCurve,
const LocHHCurve& lochhCurvejz, const LocCHCurve& locchCurvejz, const LocLHCurve& loclhCurvejz,
const LUTf& lmasklocalcurve, bool localmaskutili,
const LUTf& lmaskexplocalcurve, bool localmaskexputili,
const LUTf& lmaskSHlocalcurve, bool localmaskSHutili,
const LUTf& lmaskviblocalcurve, bool localmaskvibutili,
const LUTf& lmasktmlocalcurve, bool localmasktmutili,
LUTf& lmaskretilocalcurve, bool localmaskretiutili,
const LUTf& lmaskcblocalcurve, bool localmaskcbutili,
const LUTf& lmaskbllocalcurve, bool localmaskblutili,
const LUTf& lmaskkllocalcurve, bool localmaskklcutili,
const LUTf& lmaskloglocalcurve, bool localmasklogutili,
const LUTf& lmasklocal_curve, bool localmask_utili,
const LUTf& lmaskcielocalcurve, bool localmaskcieutili,
const LUTf& cielocalcurve, bool localcieutili,
const LUTf& cielocalcurve2, bool localcieutili2,
const LUTf& jzlocalcurve, bool localjzutili,
const LUTf& czlocalcurve, bool localczutili,
const LUTf& czjzlocalcurve, bool localczjzutili,

const LocCCmaskCurve& locccmasCurve, bool lccmasutili, const LocLLmaskCurve& locllmasCurve, bool llmasutili, const LocHHmaskCurve& lochhmasCurve, bool lhmasutili, const LocHHmaskCurve& llochhhmasCurve, bool lhhmasutili,
const LocCCmaskCurve& locccmasexpCurve, bool lccmasexputili, const LocLLmaskCurve& locllmasexpCurve, bool llmasexputili, const LocHHmaskCurve& lochhmasexpCurve, bool lhmasexputili,
const LocCCmaskCurve& locccmasSHCurve, bool lccmasSHutili, const LocLLmaskCurve& locllmasSHCurve, bool llmasSHutili, const LocHHmaskCurve& lochhmasSHCurve, bool lhmasSHutili,
const LocCCmaskCurve& locccmasvibCurve, bool lccmasvibutili, const LocLLmaskCurve& locllmasvibCurve, bool llmasvibutili, const LocHHmaskCurve& lochhmasvibCurve, bool lhmasvibutili,
const LocCCmaskCurve& locccmascbCurve, bool lccmascbutili, const LocLLmaskCurve& locllmascbCurve, bool llmascbutili, const LocHHmaskCurve& lochhmascbCurve, bool lhmascbutili,
const LocCCmaskCurve& locccmasretiCurve, bool lccmasretiutili, const LocLLmaskCurve& locllmasretiCurve, bool llmasretiutili, const LocHHmaskCurve& lochhmasretiCurve, bool lhmasretiutili,
const LocCCmaskCurve& locccmastmCurve, bool lccmastmutili, const LocLLmaskCurve& locllmastmCurve, bool llmastmutili, const LocHHmaskCurve& lochhmastmCurve, bool lhmastmutili,
const LocCCmaskCurve& locccmasblCurve, bool lccmasblutili, const LocLLmaskCurve& locllmasblCurve, bool llmasblutili, const LocHHmaskCurve& lochhmasblCurve, bool lhmasblutili,
const LocCCmaskCurve& locccmaslcCurve, bool lccmaslcutili, const LocLLmaskCurve& locllmaslcCurve, bool llmaslcutili, const LocHHmaskCurve& lochhmaslcCurve, bool lhmaslcutili,
const LocCCmaskCurve& locccmaslogCurve, bool lccmaslogutili, const LocLLmaskCurve& locllmaslogCurve, bool llmaslogutili, const LocHHmaskCurve& lochhmaslogCurve, bool lhmaslogutili,
const LocCCmaskCurve& locccmas_Curve, bool lccmas_utili, const LocLLmaskCurve& locllmas_Curve, bool llmas_utili, const LocHHmaskCurve& lochhmas_Curve, bool lhmas_utili,
const LocCCmaskCurve& locccmascieCurve, bool lccmascieutili, const LocLLmaskCurve& locllmascieCurve, bool llmascieutili, const LocHHmaskCurve& lochhmascieCurve, bool lhmascieutili,

const LocHHmaskCurve& lochhhmas_Curve, bool lhhmas_utili,
const LocwavCurve& loclmasCurveblwav, bool lmasutiliblvav,
const LocwavCurve& loclmasCurvecolwav, bool lmasutilicolwav,
const LocwavCurve& locwavCurve, bool locwavutili,
const LocwavCurve& locwavCurvejz, bool locwavutilijz,
const LocwavCurve& loclevwavCurve, bool loclevwavutili,
const LocwavCurve& locconwavCurve, bool locconwavutili,
const LocwavCurve& loccompwavCurve, bool loccompwavutili,
```



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

currying

category theory

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** / **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**

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

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

Imperative

HOW



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

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

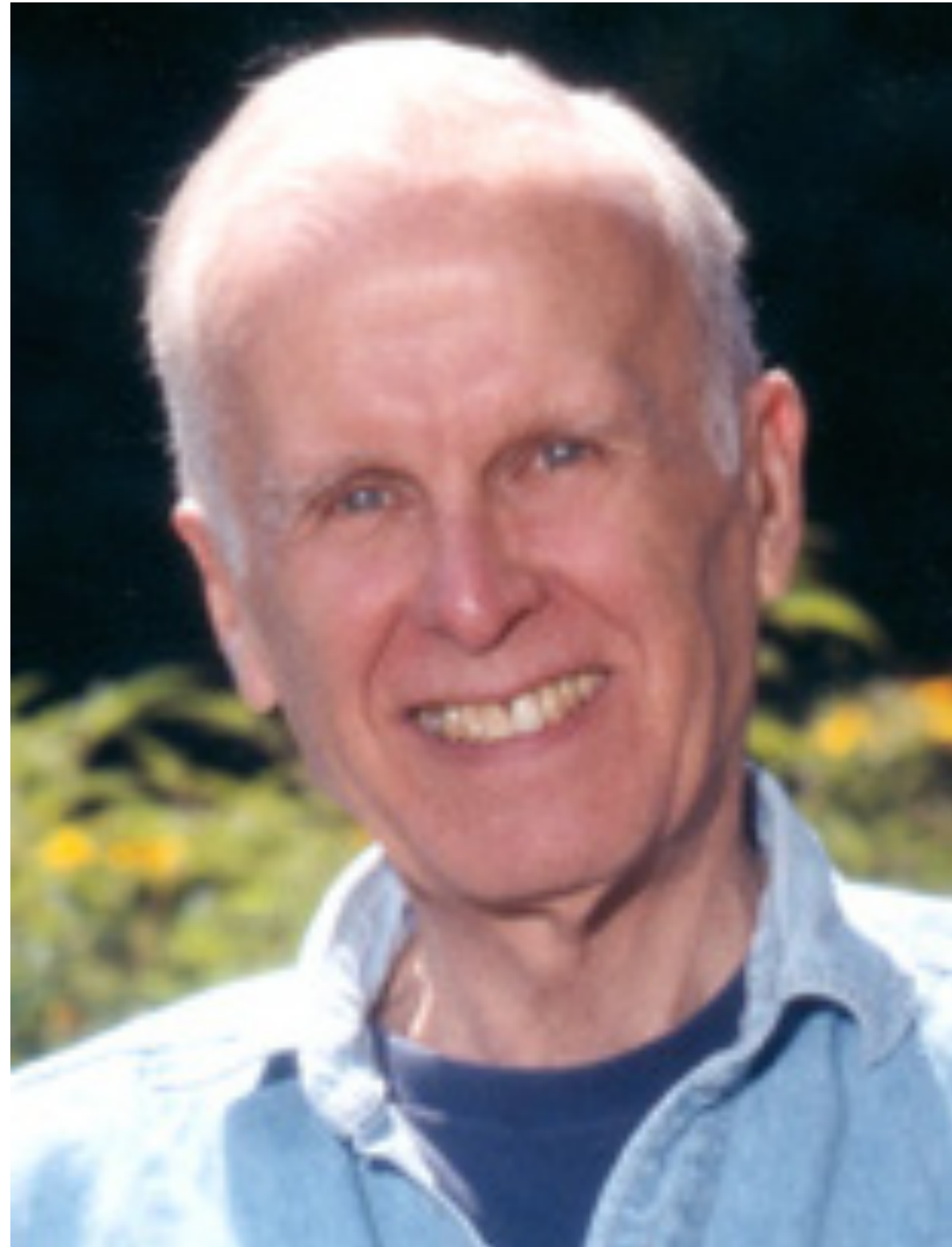
1960s



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

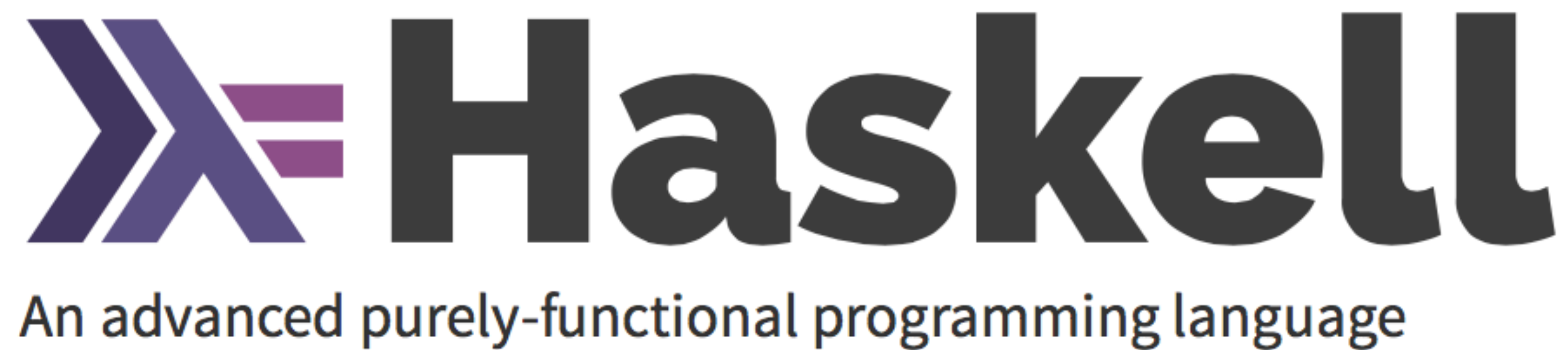
1970-80s



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

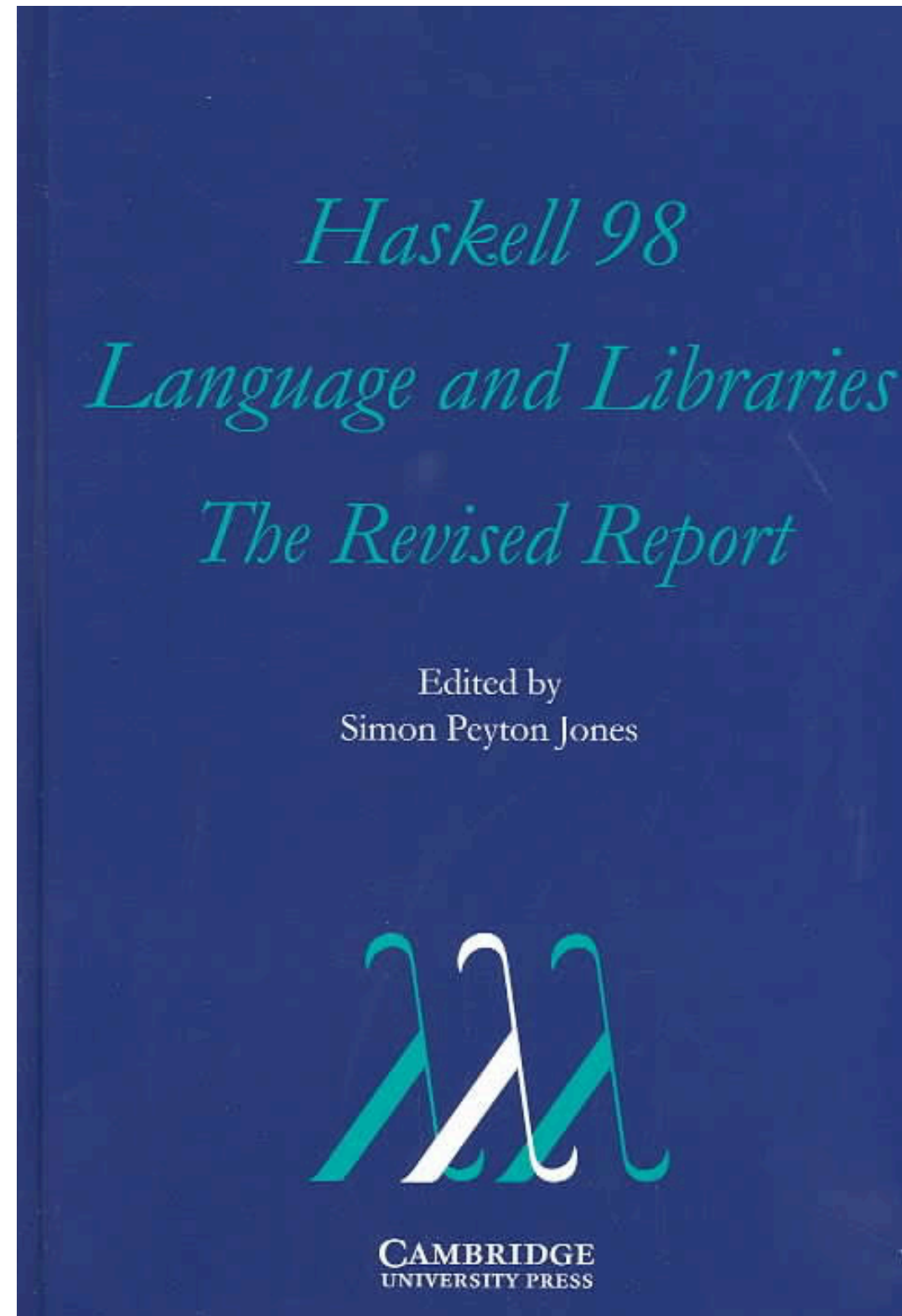
1990s



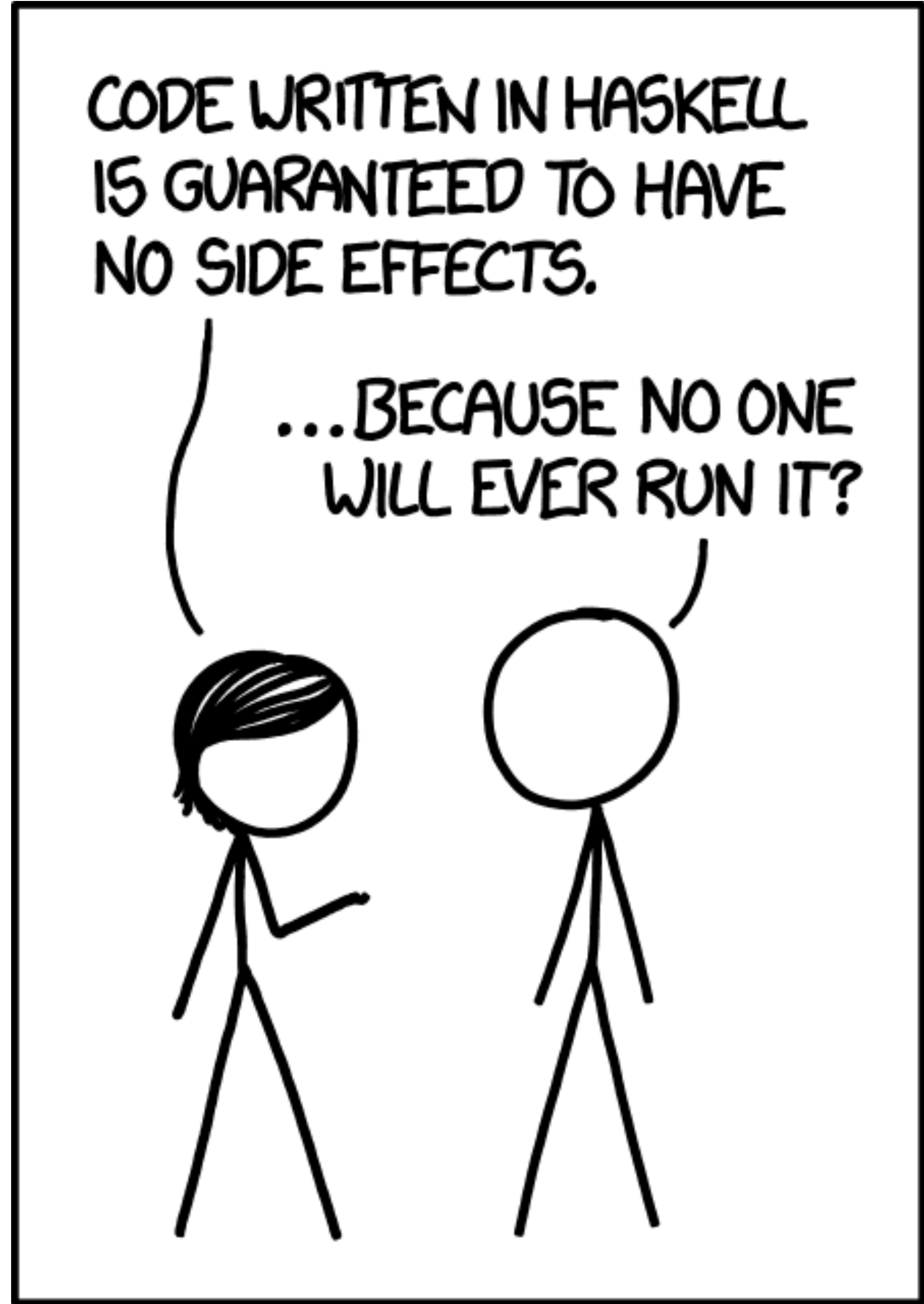
Phil Wadler and others develop **type classes** and **monads**,
two of the main innovations of Haskell

Historical Background

2003



The committee publishes the **Haskell Report**, defining a **stable** version of the language



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](#).

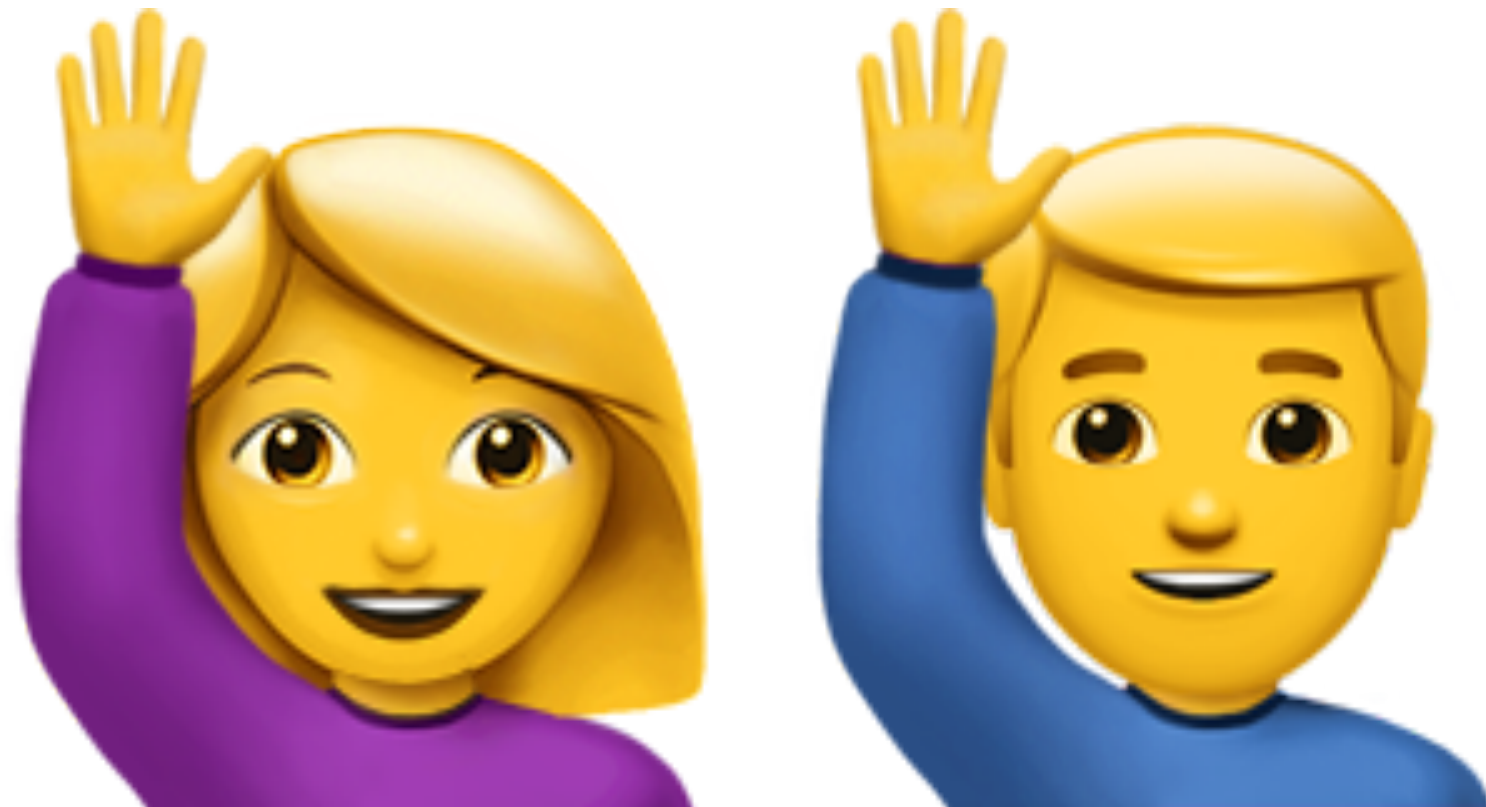
Modern C++ is functional

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

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


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);

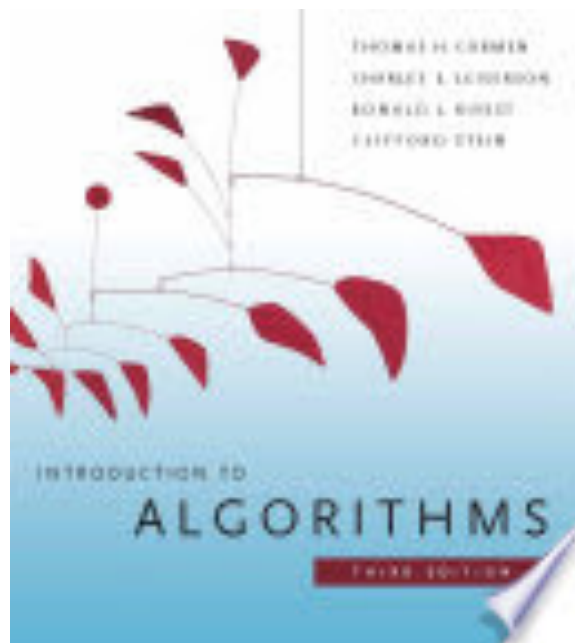
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
/* 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

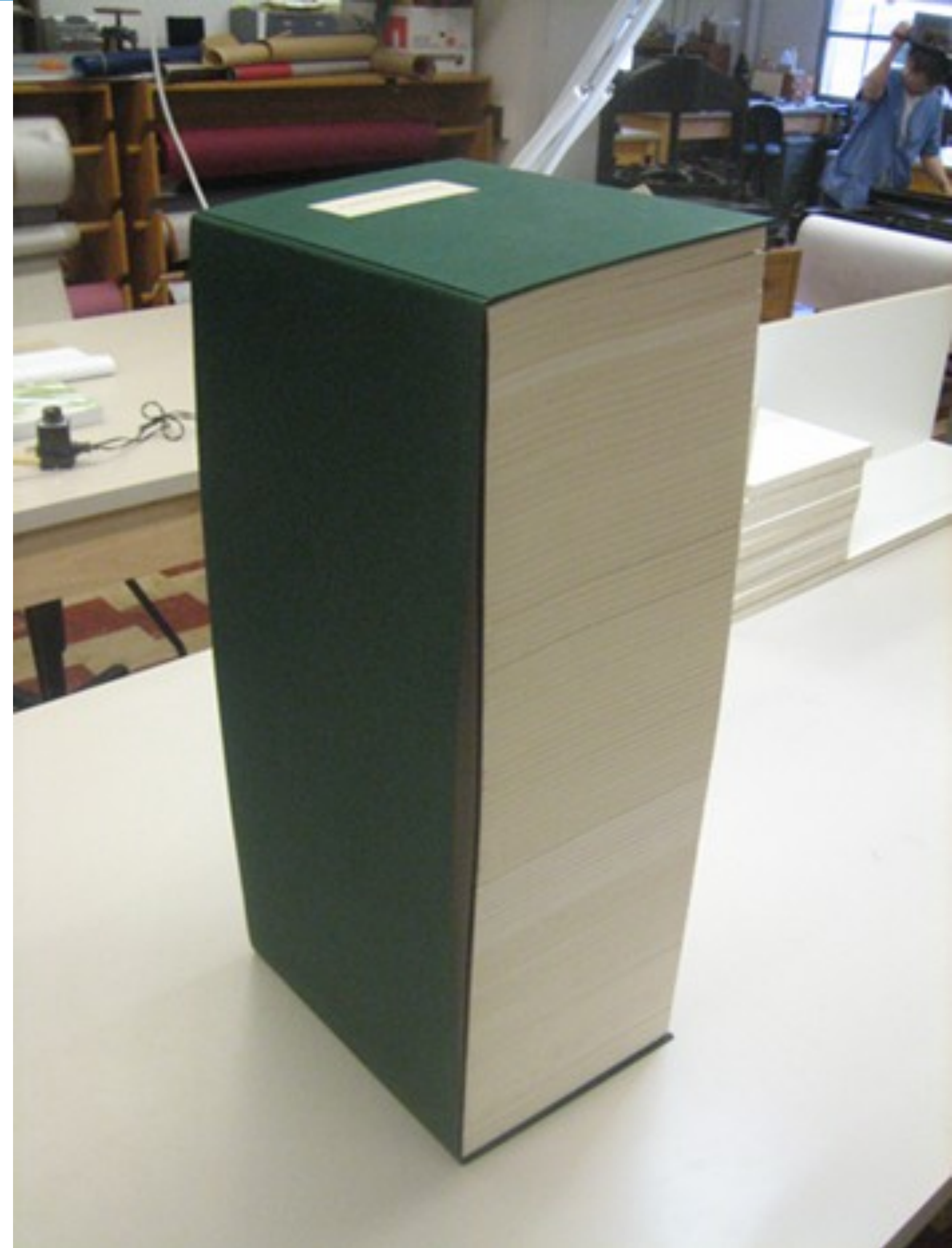
It's all about | pipelines

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



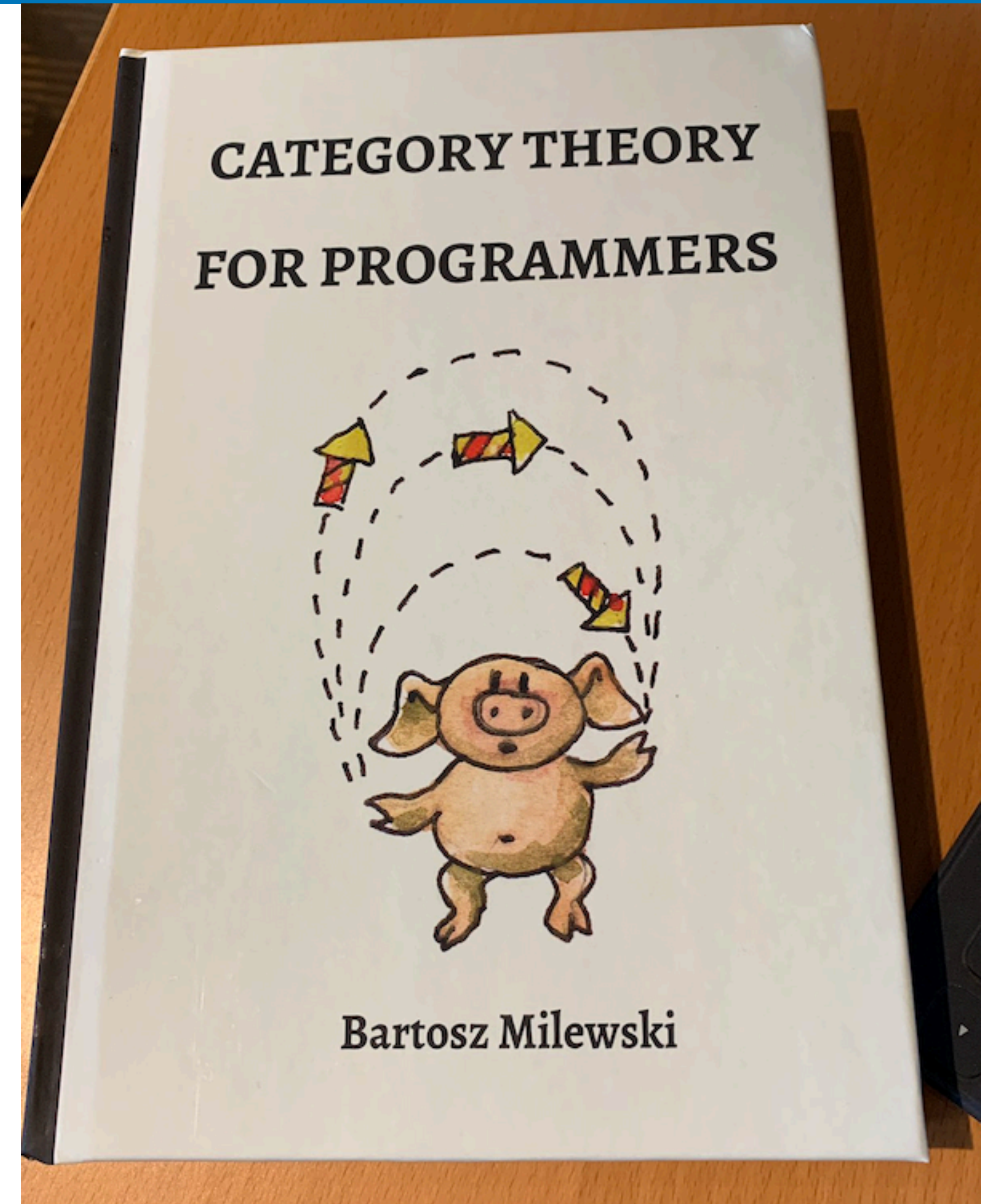
The Book

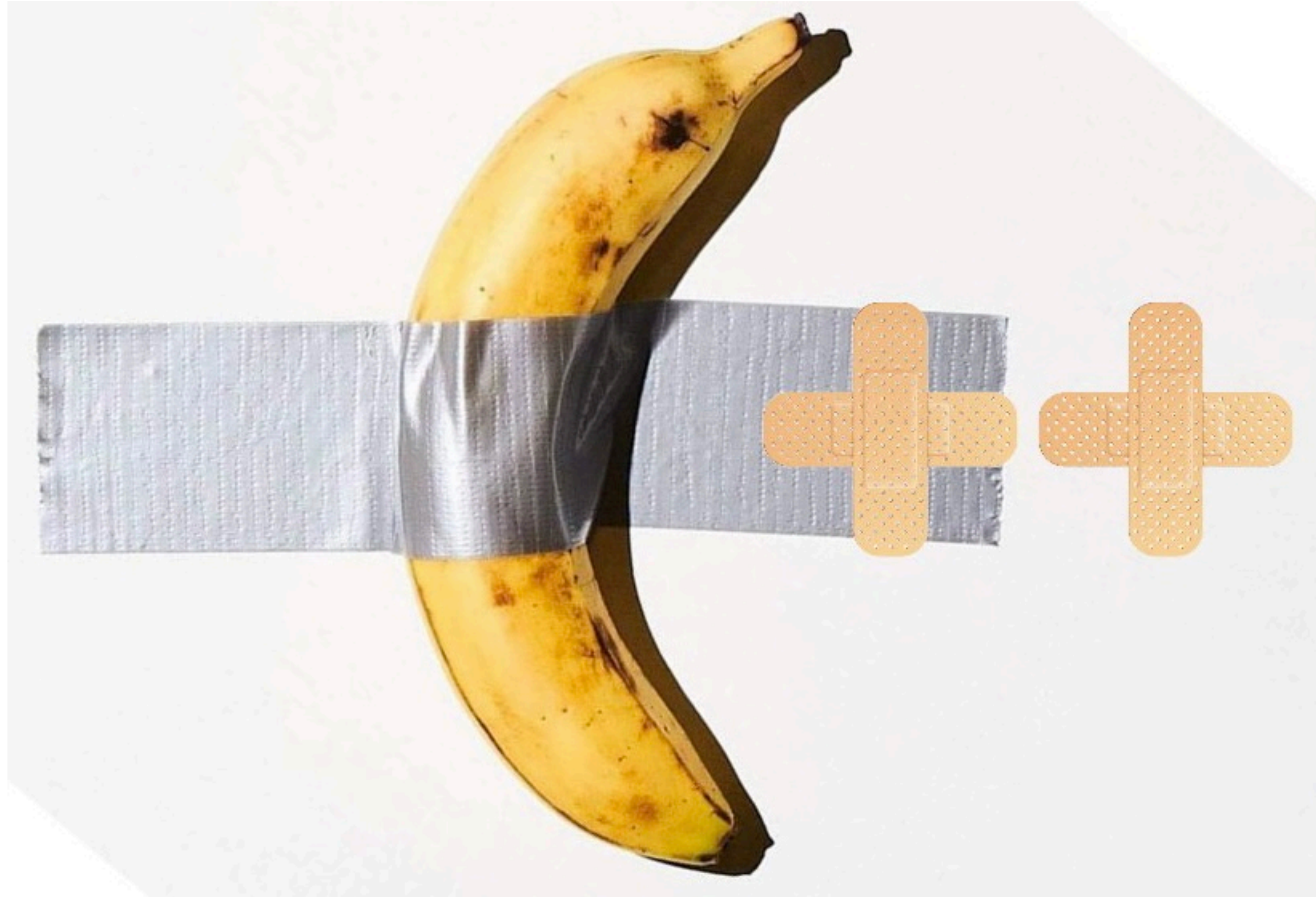


Bartosz Milewski

@BartoszMilewski

github.com/hmemcpy/milewski-ctfp-pdf





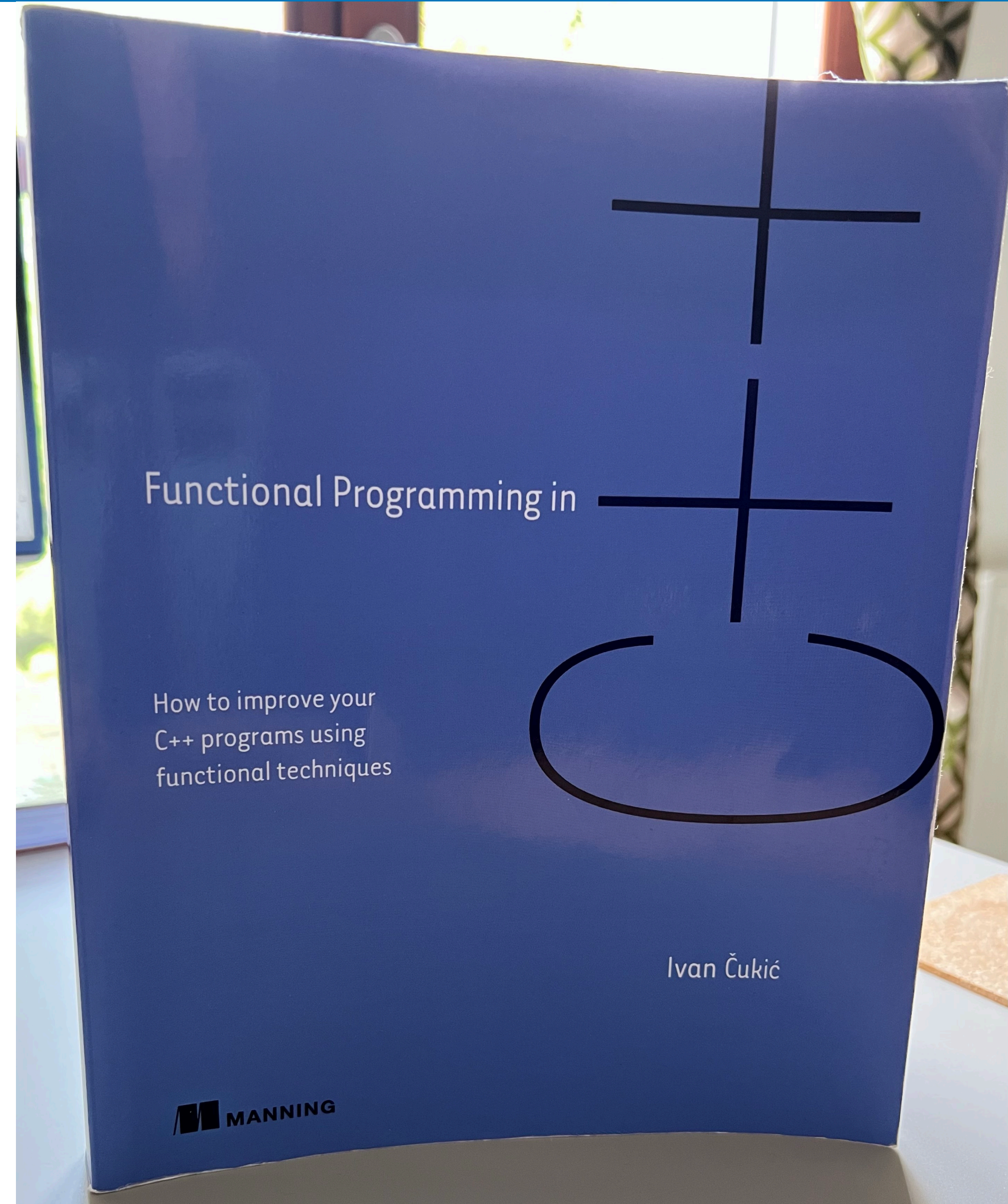
twitter.com/tvaneerd/status/1387

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 '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
```

Higher Order Functions for Ordinary C++ Developers

Björn Fähler

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

Higher Order Functions – Meeting C++ 2018 © Björn Fähler

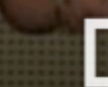
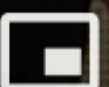


@bjorn_fahller

1/93

Meeting C++ 2018
Björn Fähler
Higher Order Functions
for Ordinary C++
Developers

0:14 / 52:28



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

youtube.com/watch?v=qL6zUn7iiLg



Boxes

Type Constructors

There are various ways to hide  a value:

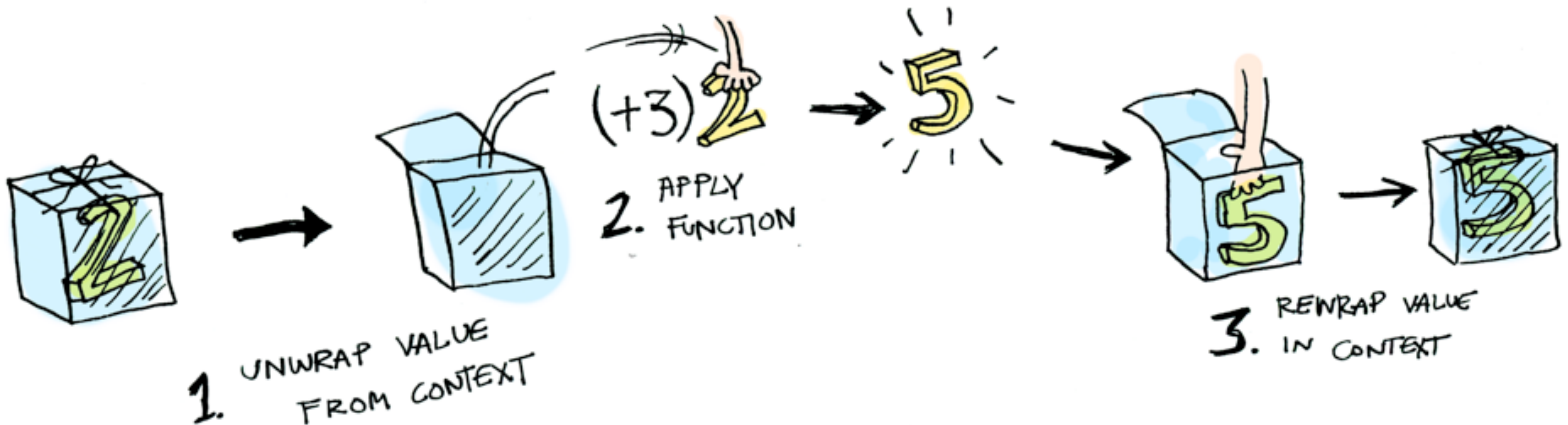
- `unique_ptr<T> p;`
- `shared_ptr<T> p;`
- `vector<T> v;`
- `optional<T> o;`
- `function<T(int)> f;`

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

`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>`)





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

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

if / else

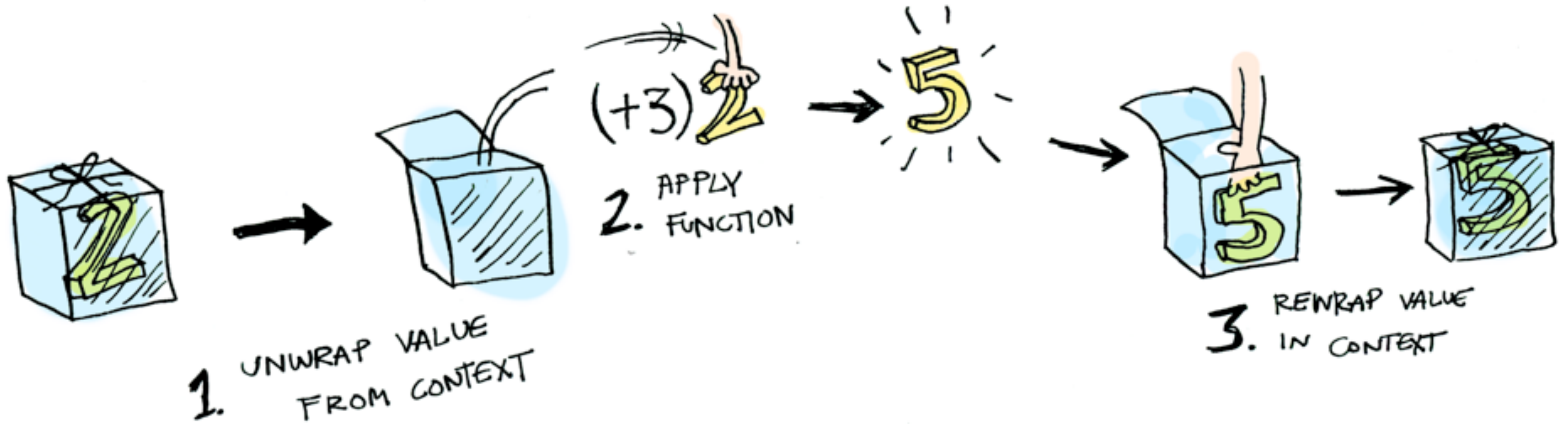
```
optional<T> g(optional<T> in)
```

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);
```

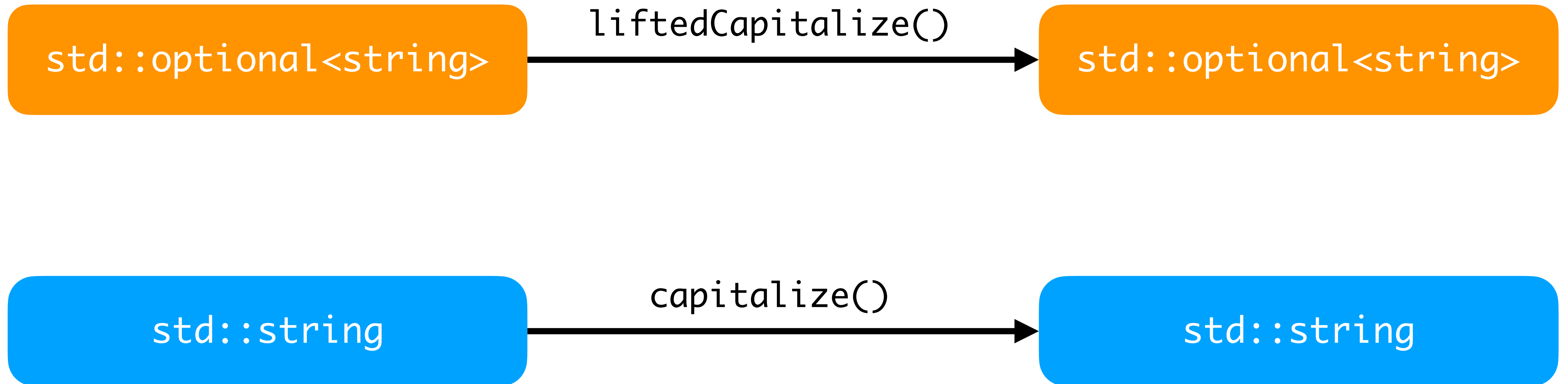
Lifting `capitalize()`

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

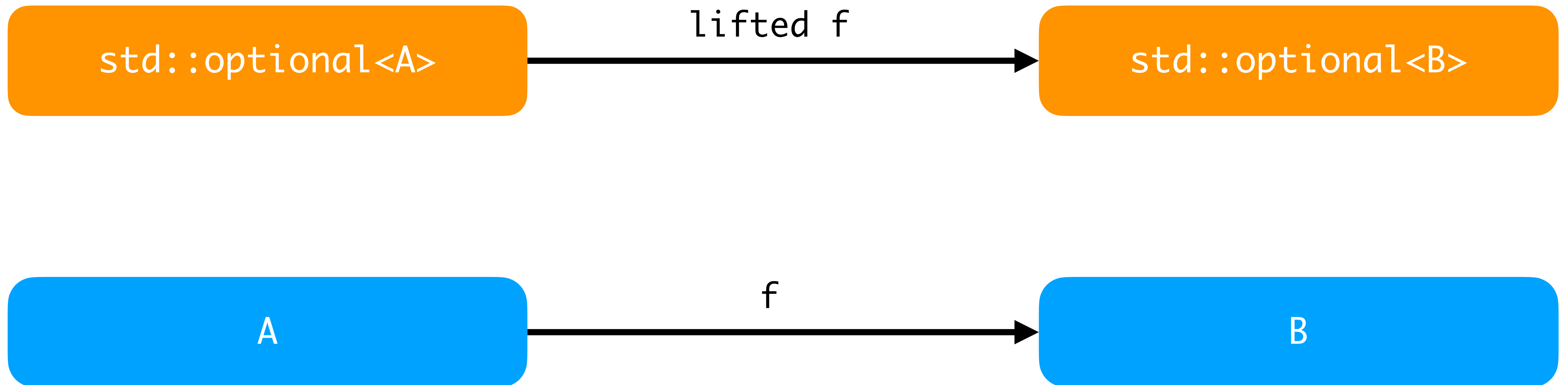
```
optional<string> liftedCapitalize(const optional<string> & s)
{
    optional<string> result;
    if (s)
        result = capitalize(*s);

    return result;
}
```


Lifting capitalize()



Lifting any function



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 (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`

```
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;
}
```

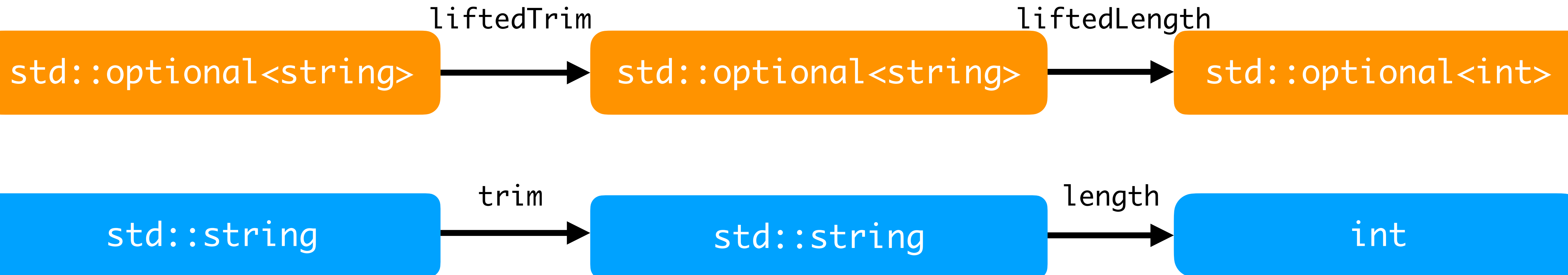
```
vector<string> names{ ... };
```

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

Why?

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

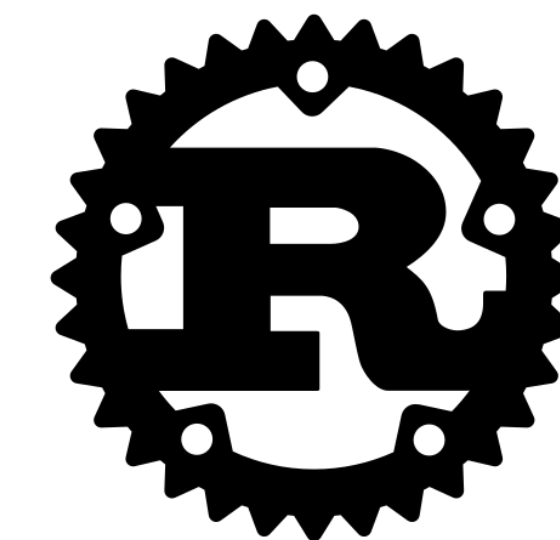
Monadic `std::optional` (C++23 P0798)

```
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()
```

```
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

```
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
```





`transform()`

`and_then()`

functor

monad



`fmap`

`>>= (bind)`

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>`

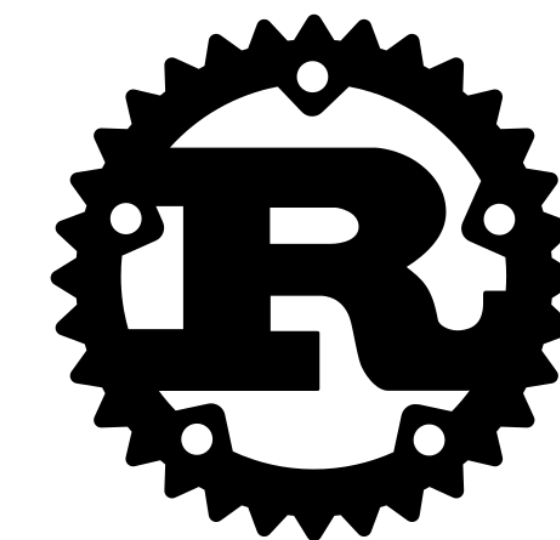
either the expected **T** value

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



```
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),  
    }  
}
```

#[derive(Debug)]
struct DivisionByZero;

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

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

Result?

Heritage

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

```
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
```

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



std::optional

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

C++ 17

`std::expected`

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

C++ 23

.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 N/A
- Microsoft STL VS2022 17.6

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




Sy Brand

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

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

Expressions yield values, Statements do not;

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

The image shows a YouTube video player interface. At the top left, there is a logo for Cppcon 2022, 'The C++ Conference', with the dates 'September 12th-16th'. The main video area shows a man, Dave Abrahams, speaking at a podium. To the right of the video, the title 'Values' is displayed in yellow, followed by the subtitle 'Regularity, Independence, Projection, and the Future of Programming' in white. Below the subtitle, the speaker's name 'DAVE ABRAHAMS' is shown in yellow. At the bottom of the video area, there is a Cppcon logo and the dates '2022 | September 12th-16th'. The video player controls at the bottom include a play button, a progress bar showing '0:10 / 1:00:45', and a 'Introduction' chapter marker. There are also icons for closed captions, settings, HD, and other video controls. A 'think-cell' logo is visible in the bottom right corner of the video area.

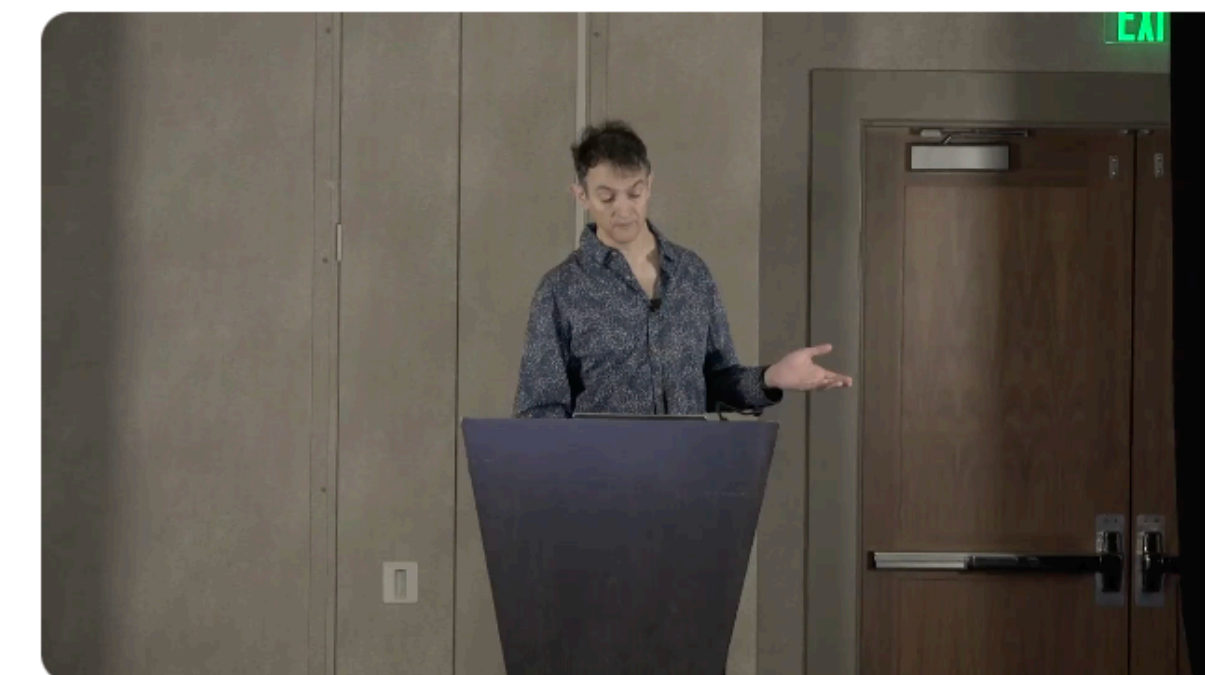
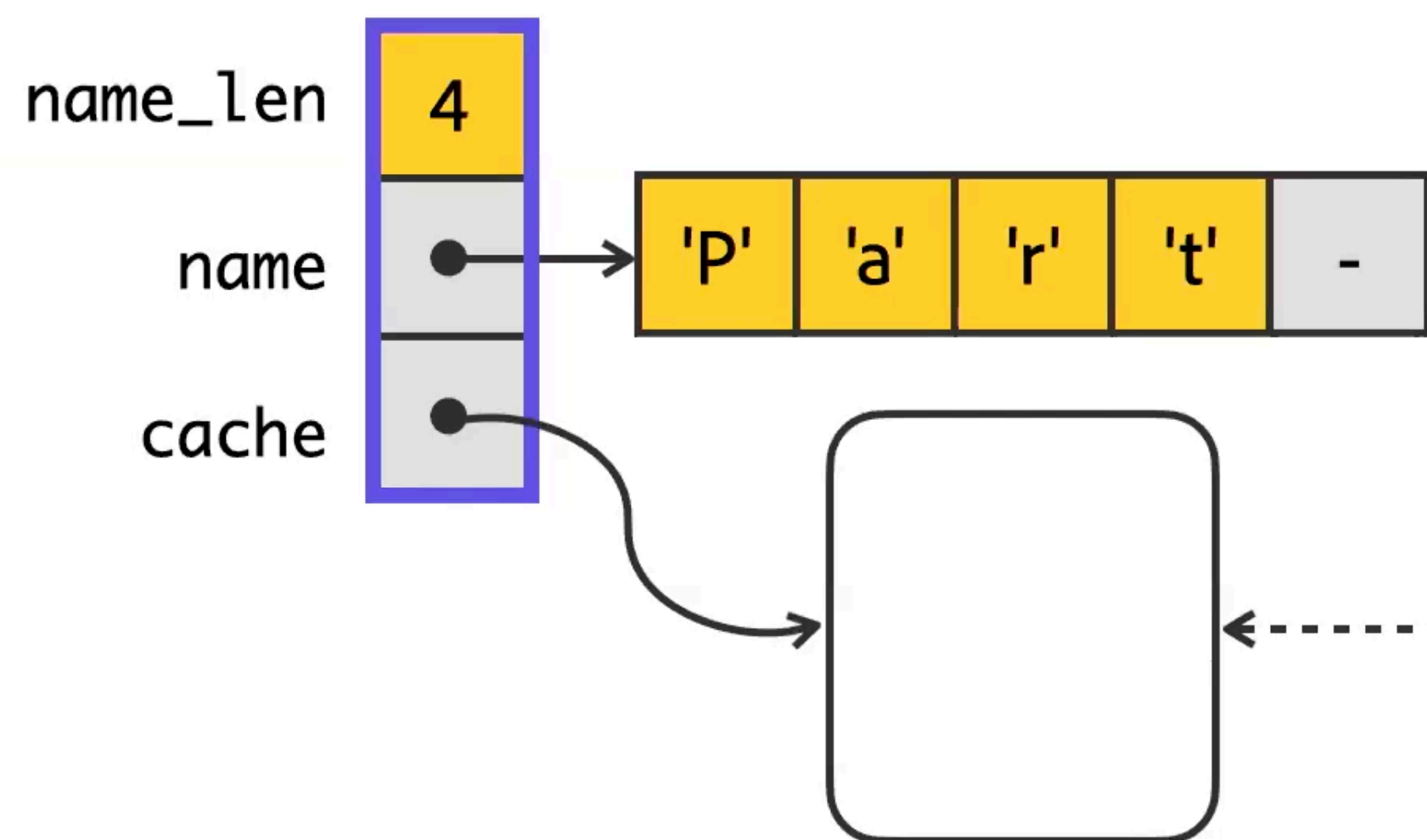
Value Semantics: Safety, Independence, Projection, & Future of Programming - Dave Abrahams CppCon 22

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

The image shows a video player interface. The main video area displays the text "value semantics" in large white font on a dark background. 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 video area, there is a smaller video thumbnail showing a man (Juan Pedro Bolívar Puente) speaking, with his name "JUAN PEDRO BOLÍVAR PUENTE" written below it. Below the thumbnail, the title "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, a pause button, and a settings menu with "HD" and "cppcon.org" labels.

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

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

RADIX BALANCED SEARCH

v[17] → 01 00 01

a b c d e f g h i j k l m n o p q r s t u v y

JUAN PEDRO BOLIVAR PUENTE

Postmodern immutable data structures

0:00 / 1:05:59

CC HD CppCon.org

CppCon 2017

CppCon 2017: Juan Pedro Bolivar Puente “Postmodern immutable data structures”

youtube.com/watch?v=sPhpelUfu8Q

C++ 20 Ranges

The beginning of the end for [begin, end)

Jeff Garland



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

Conor Hoekstra's Top 10 algorithms problems: Sushi for 2 🍣

godbolt.org/z/oras9sEE3

```
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;
}
```



Tristan Brindle
@tristanbrindle

github.com/tcbrindle/flux

```
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();
}
```

```
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_));
}
```


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 =  
    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 / views*

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

* the Nico slide :)

Remember him?



Phil Wadler and others develop **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

The Imperatives Must Go!



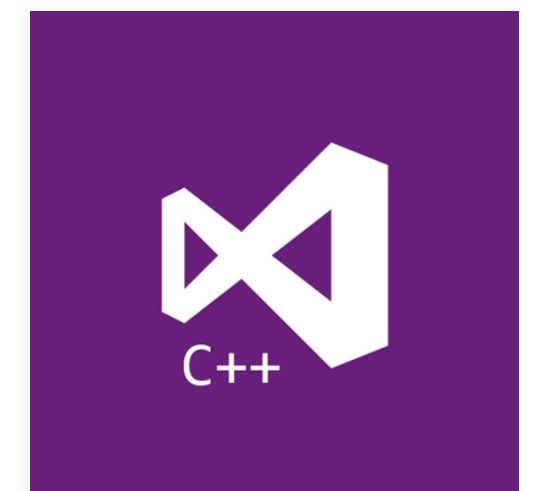
Core C++

June 2023

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

 [@ciura_victor@hachyderm.io](mailto:ciura_victor@hachyderm.io)

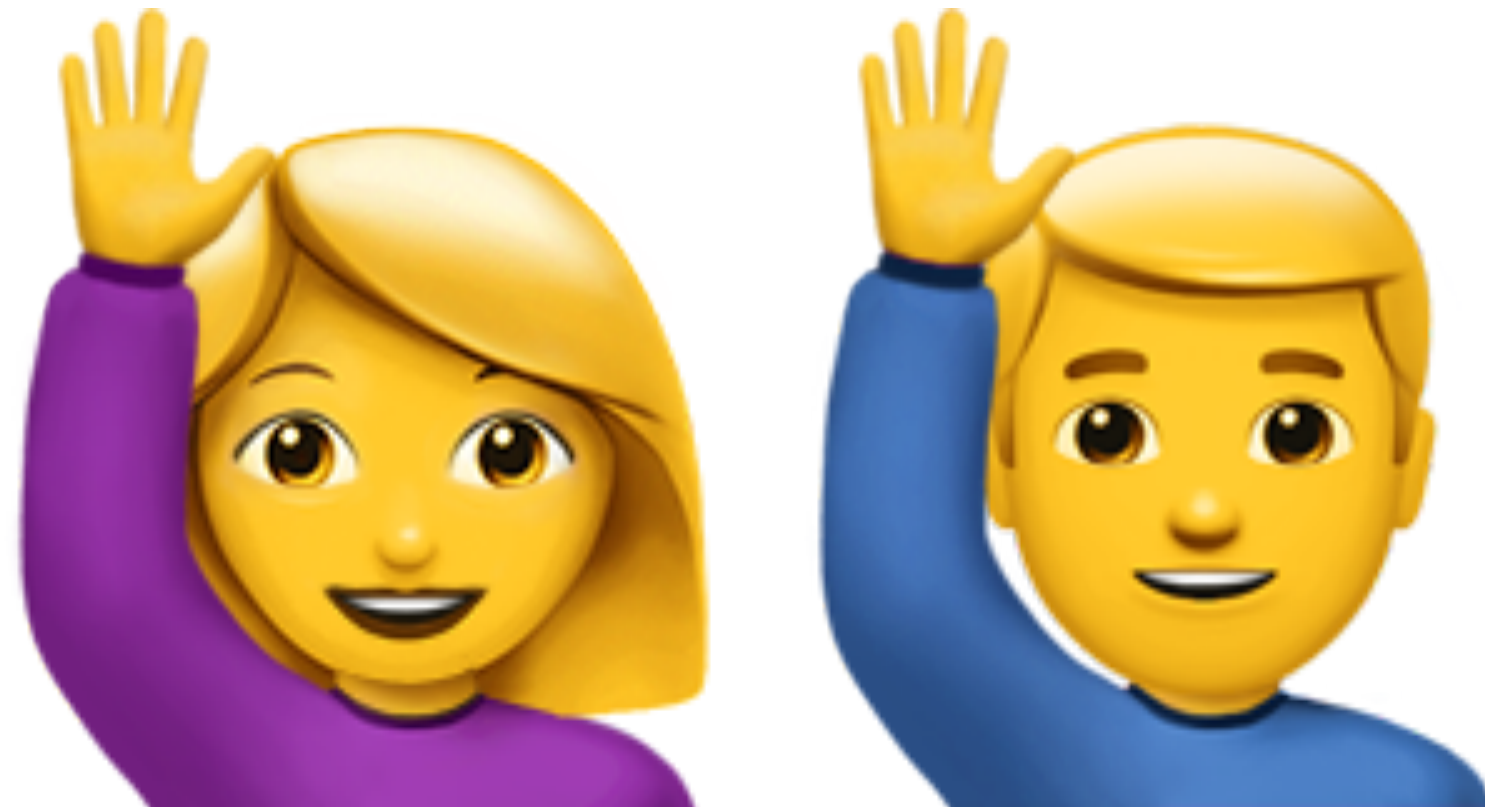
Victor Ciura
Principal Engineer
Visual C++



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

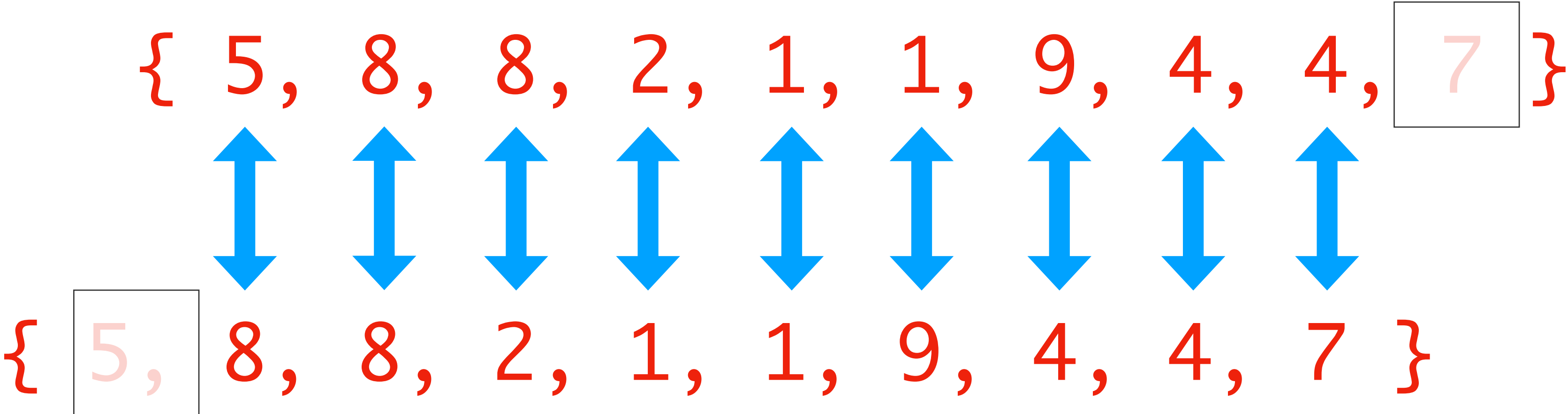
Visual hint:

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

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

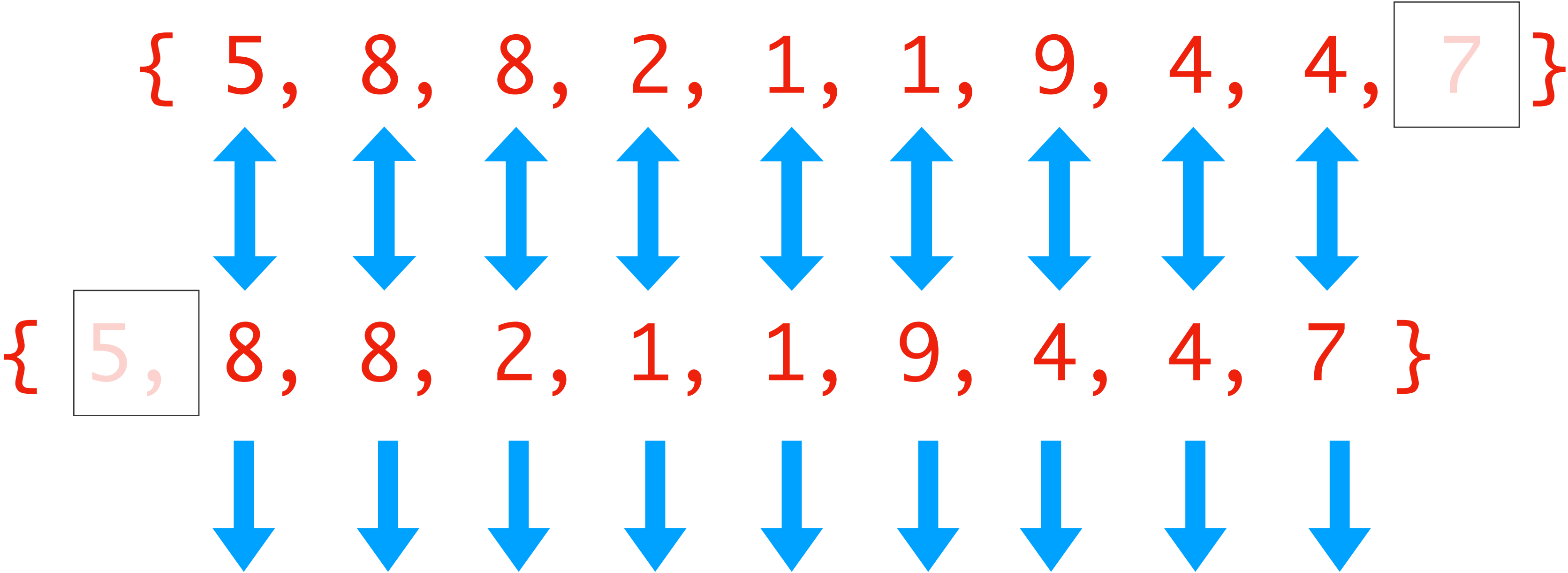
Counting adjacent repeated values in a sequence

Visual hint:



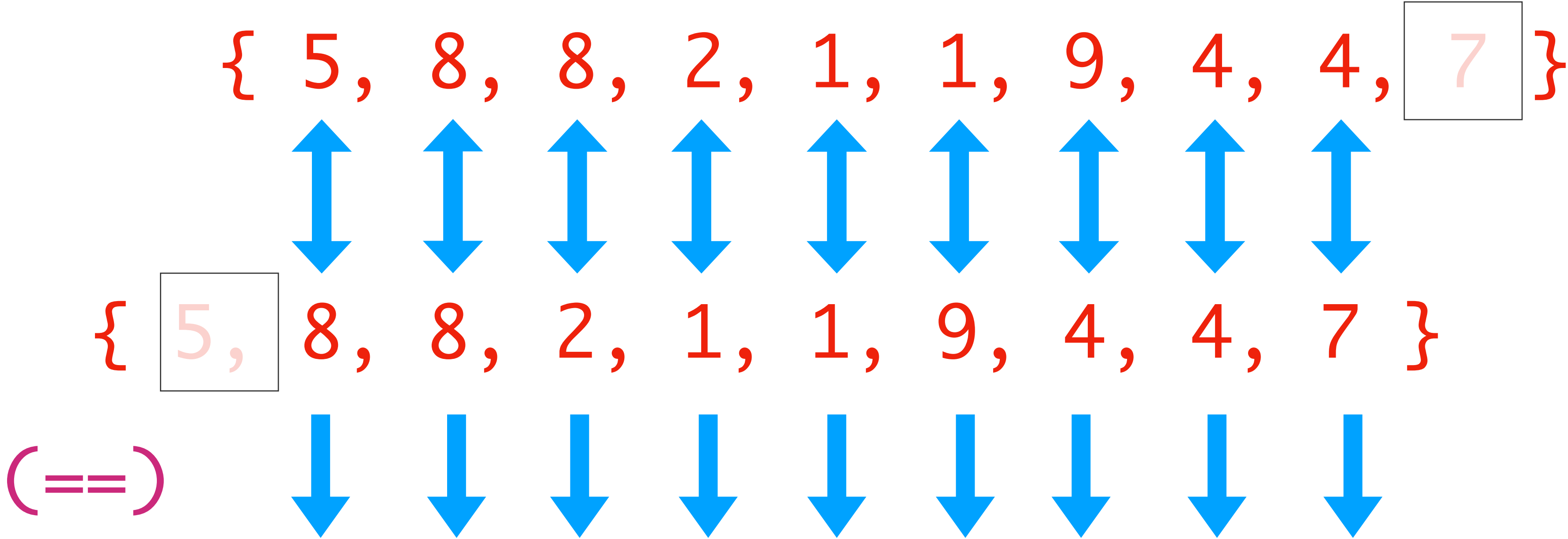
Counting adjacent repeated values in a sequence

Visual hint:



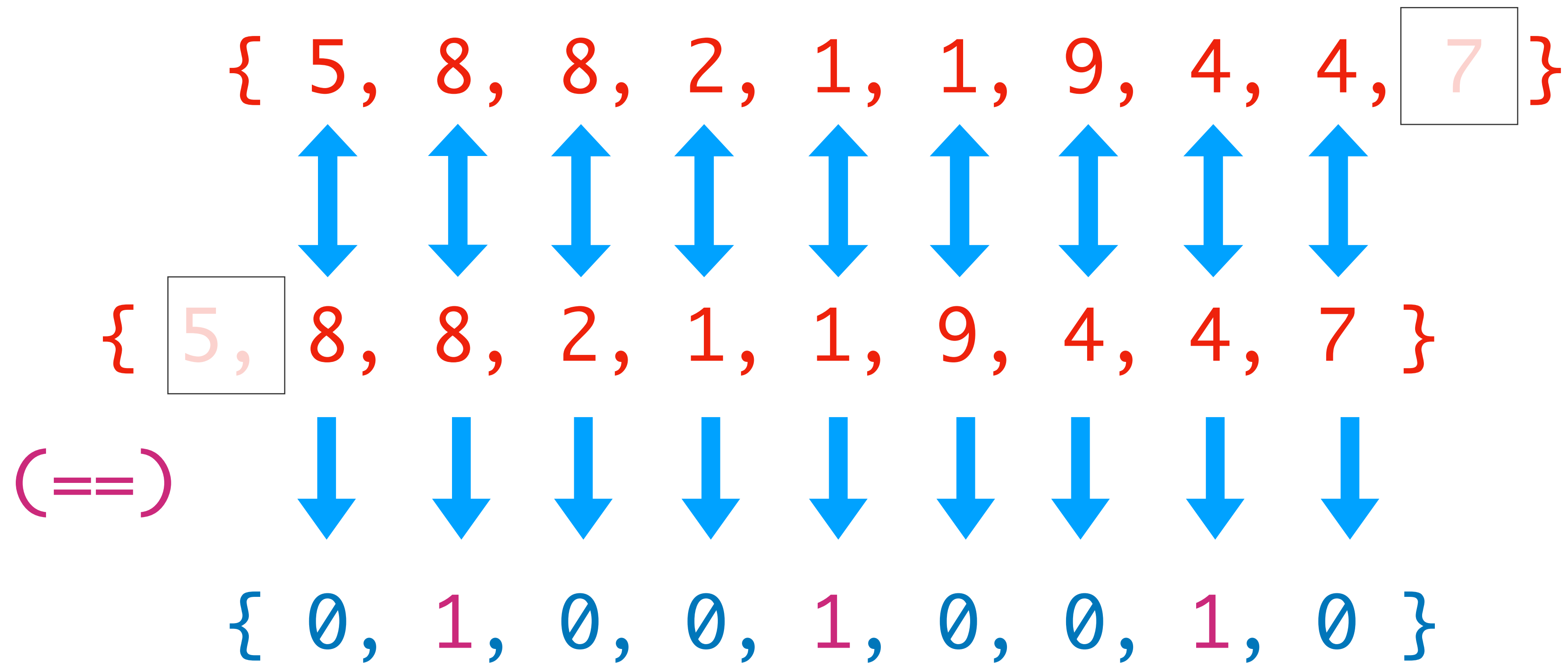
Counting adjacent repeated values in a sequence

Visual hint:



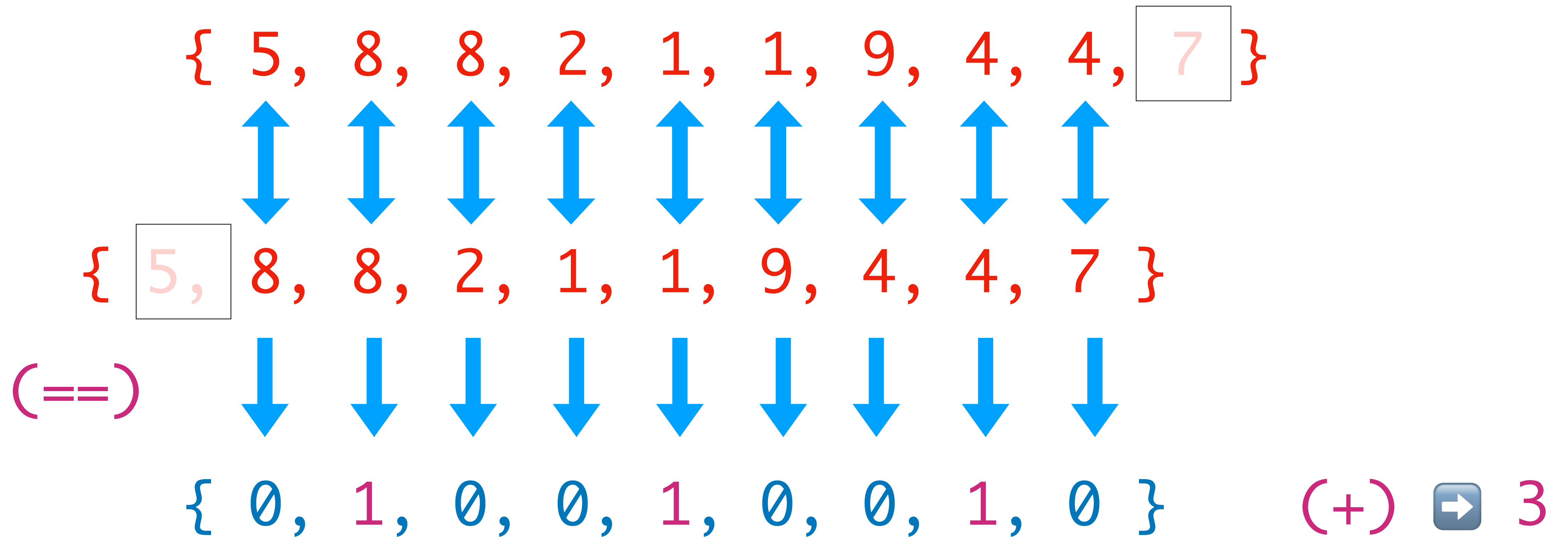
Counting adjacent repeated values in a sequence

Visual hint:



Counting adjacent repeated values in a sequence

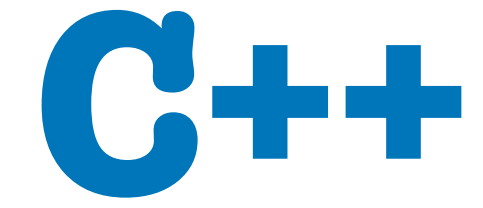
Visual hint:



C++

Counting adjacent repeated values in a sequence

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

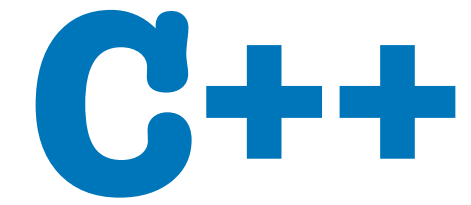


Counting adjacent repeated values in a sequence

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

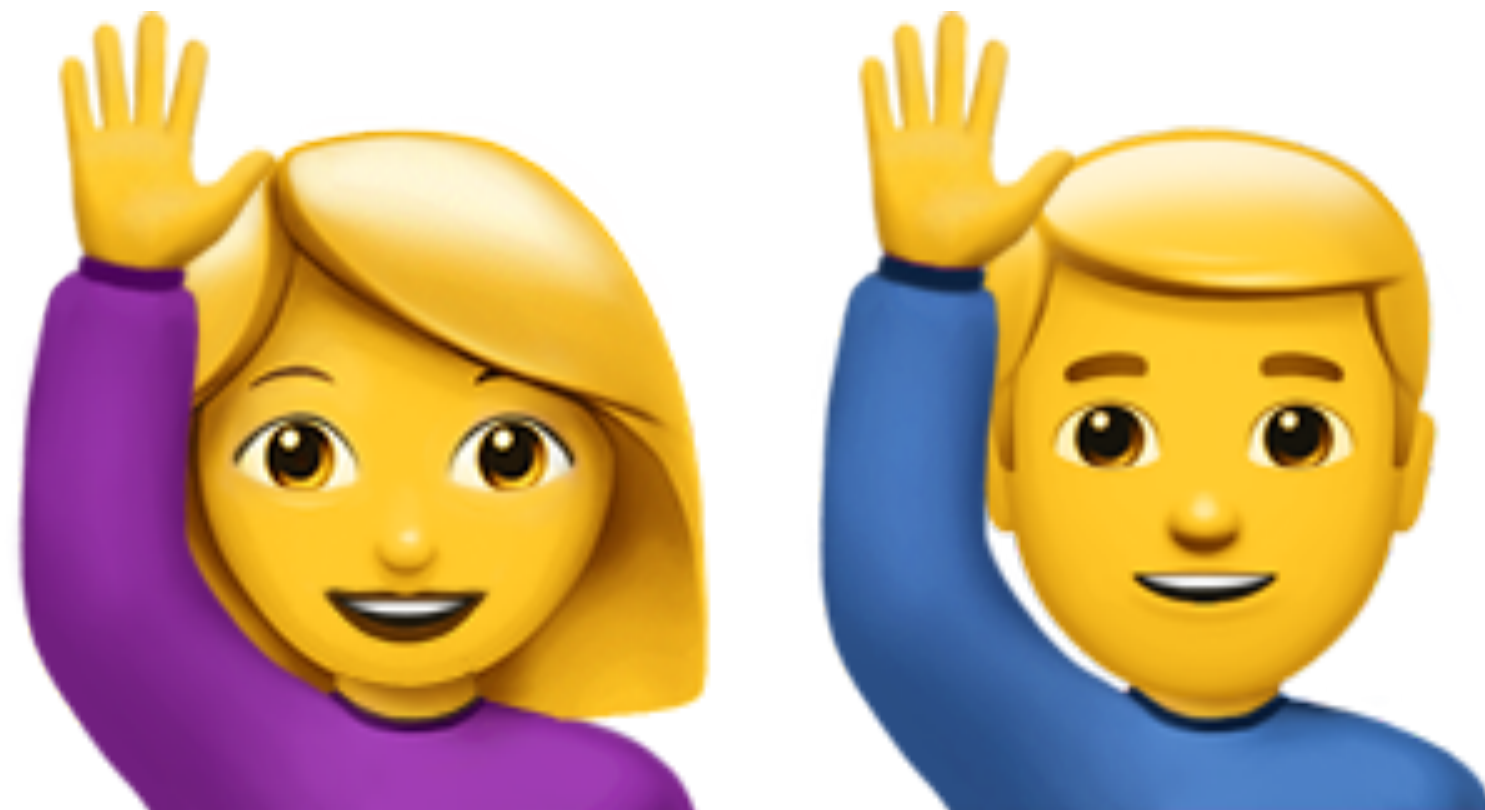
```
template <typename T>
int count_adj_equals(const T & xs) // requires non-empty range
{
    return std::inner_product(
        std::cbegin(xs), std::cend(xs) - 1, // to penultimate elem
        std::cbegin(xs) + 1,                // collection tail
        0,
        std::plus{},
        std::equal_to{});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

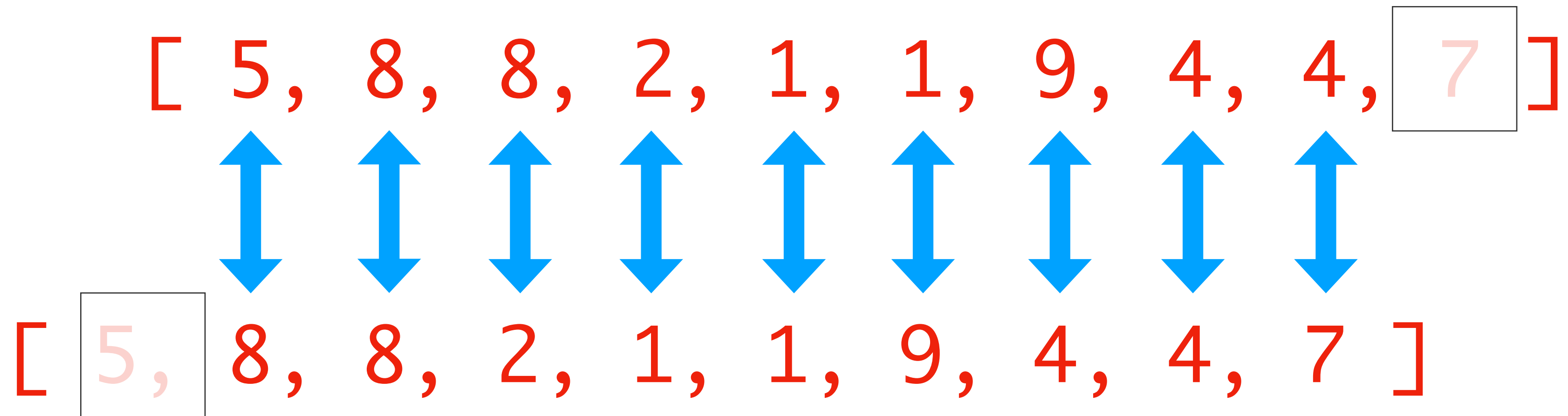
[5, 8, 8, 2, 1, 1, 9, 4, 4, 7]

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



Counting adjacent repeated values in a sequence

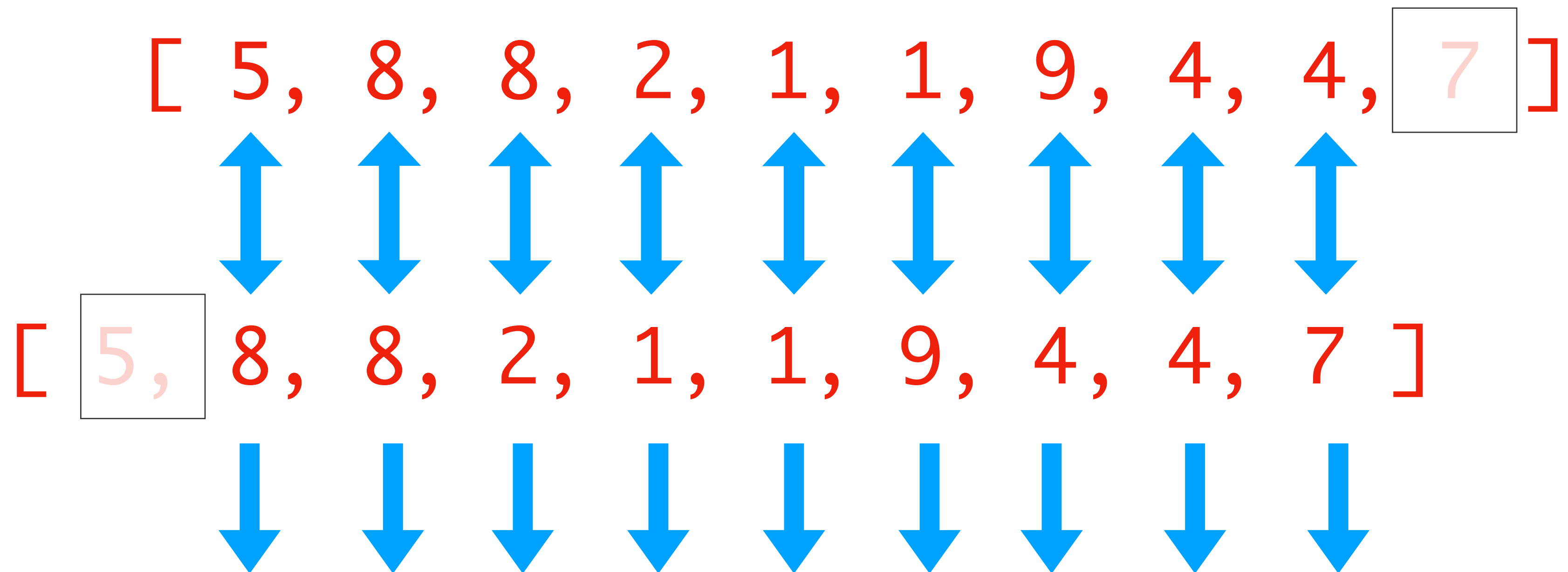
Visual hint:





Counting adjacent repeated values in a sequence

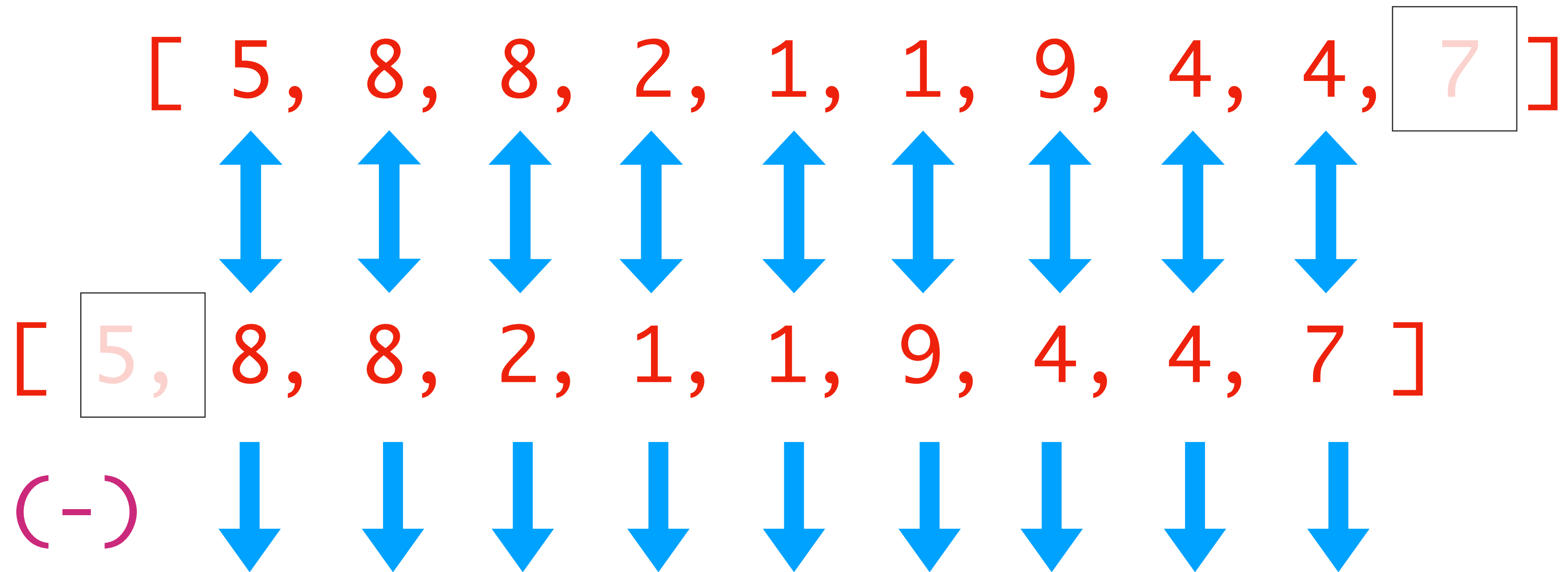
Visual hint:





Counting adjacent repeated values in a sequence

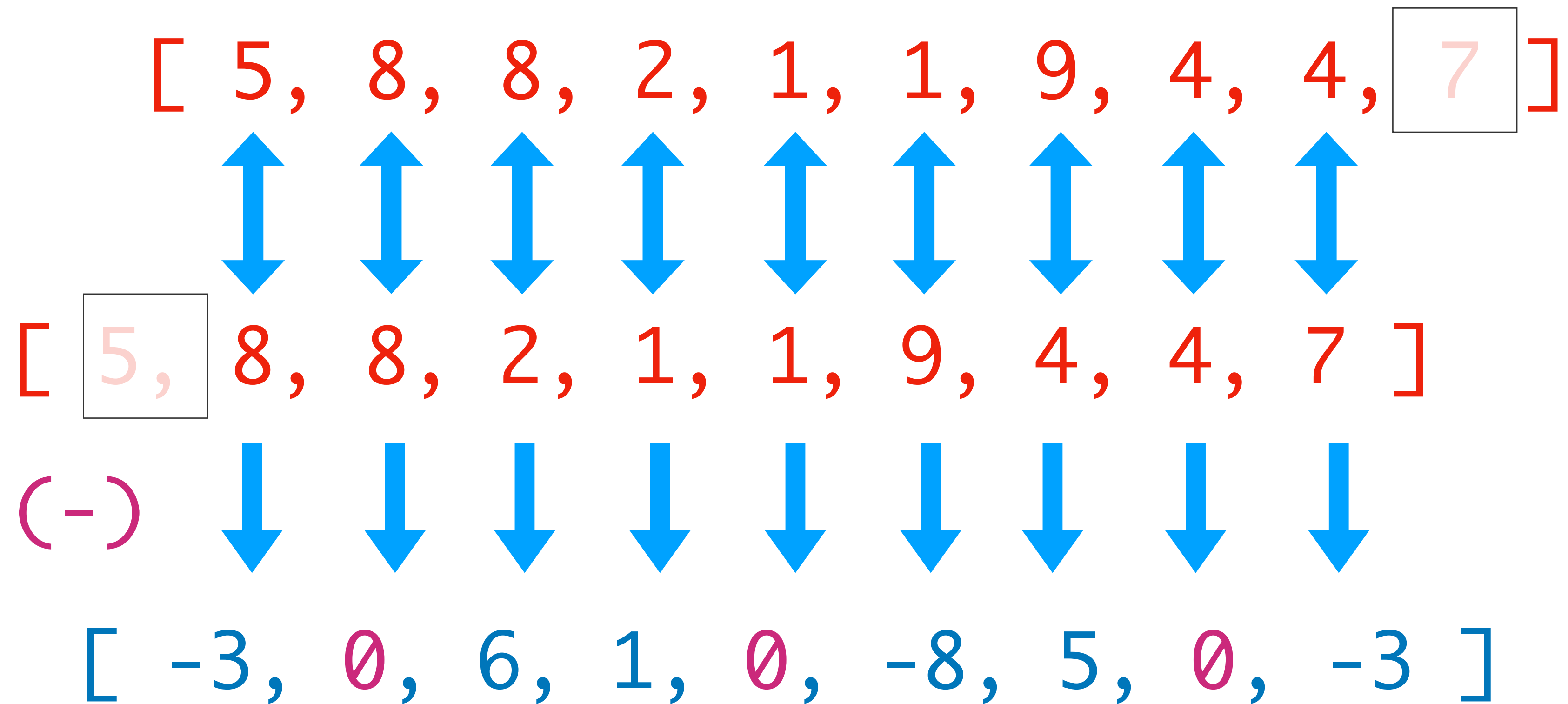
Visual hint:





Counting adjacent repeated values in a sequence

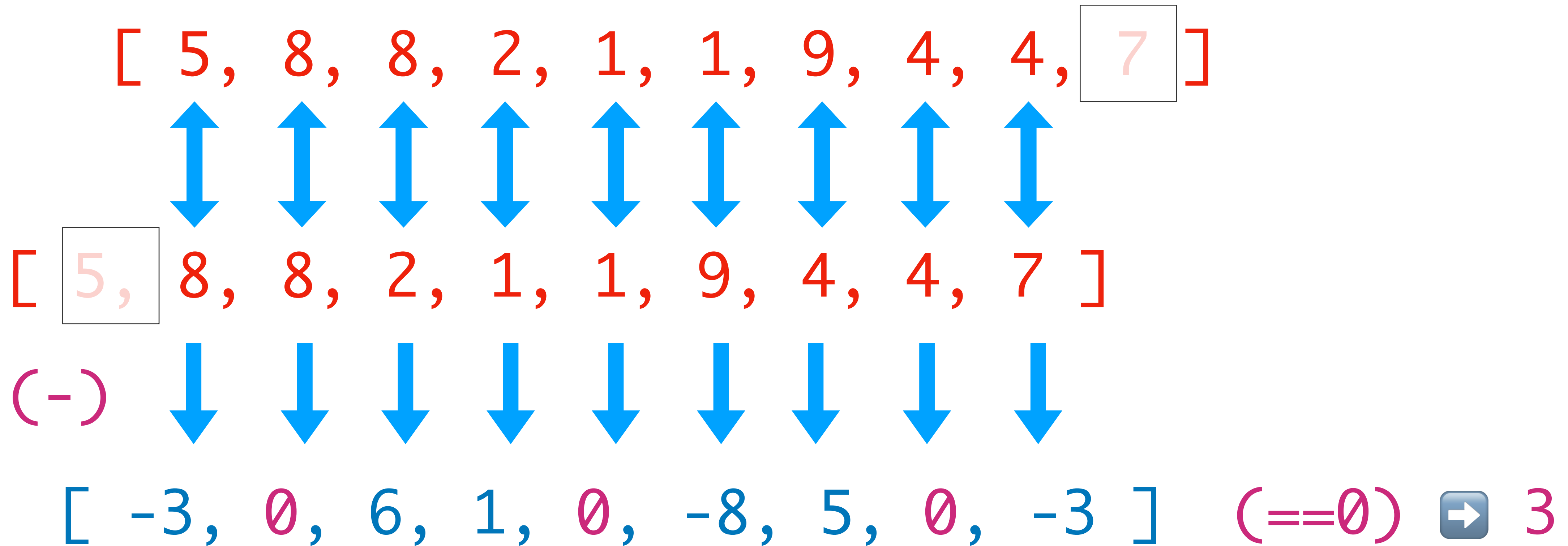
Visual hint:





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