accu
2023

AND THEN() SOME(T)

VICTOR CIURA

And Then() Some(T)

ACCU

April 2023

🐦 @ciura_victor
🐘 @ciura_victor@hachyderm.io

Victor Ciura
Principal Engineer
Visual C++

Abstract

Don't look in the box!

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

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

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

About me

Advanced Installer

Clang Power Tools

Visual C++

@ciura_victor

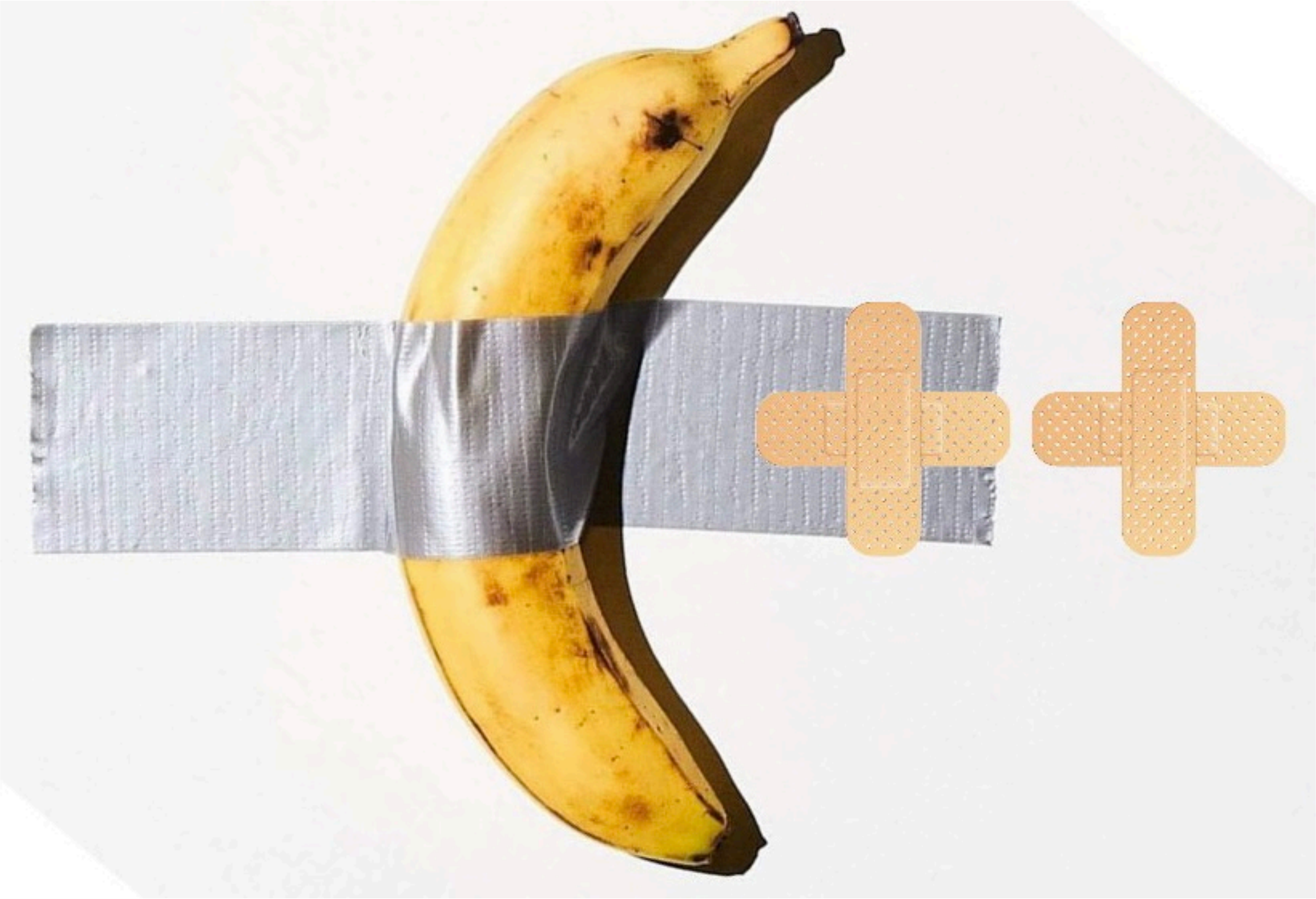
Modern C++ is functional

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 ...

Modern C++



twitter.com/tvaneerd/status/1387

Boxes 📦

Type Constructors

There are various ways to hide 📦 a **value**:

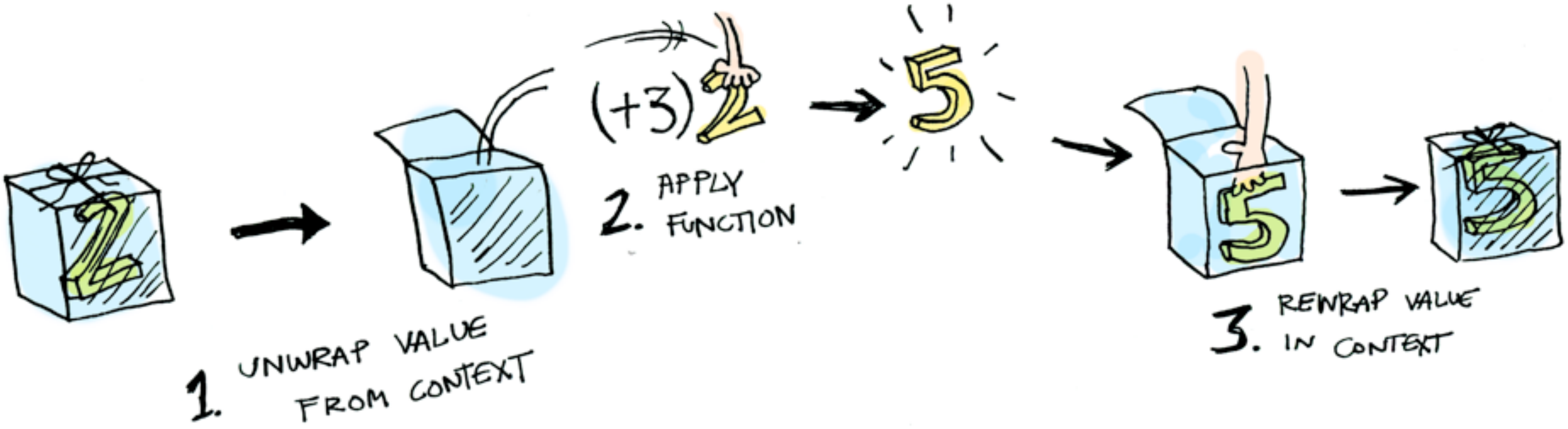
- `unique_ptr<T> p;`
- `shared_ptr<T> p;`
- `vector<T> v;`
- `optional<T> o;`
- `function<T(int)> f;`

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

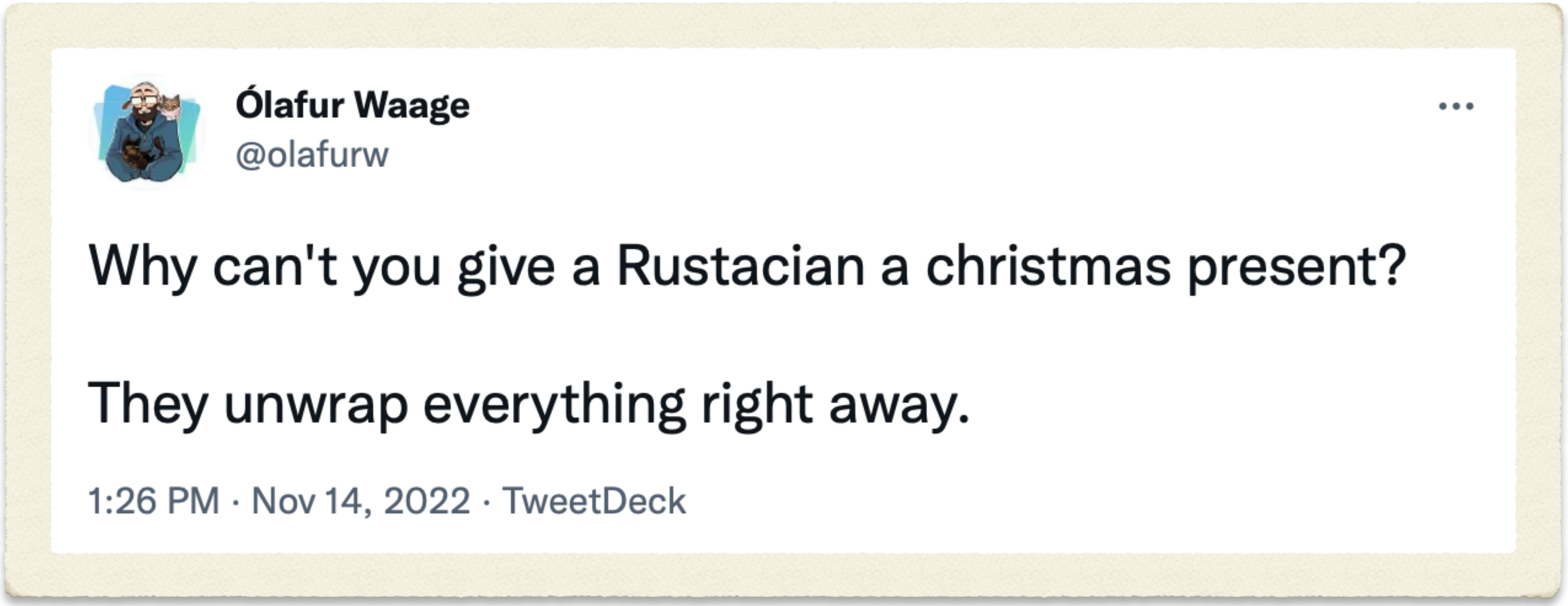
`std::optional` can simplify code

- 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>`)

fmap (+3)
2
SOME MAGIC HAPPENS
5

The Box



Ólafur Waage
@olafurw

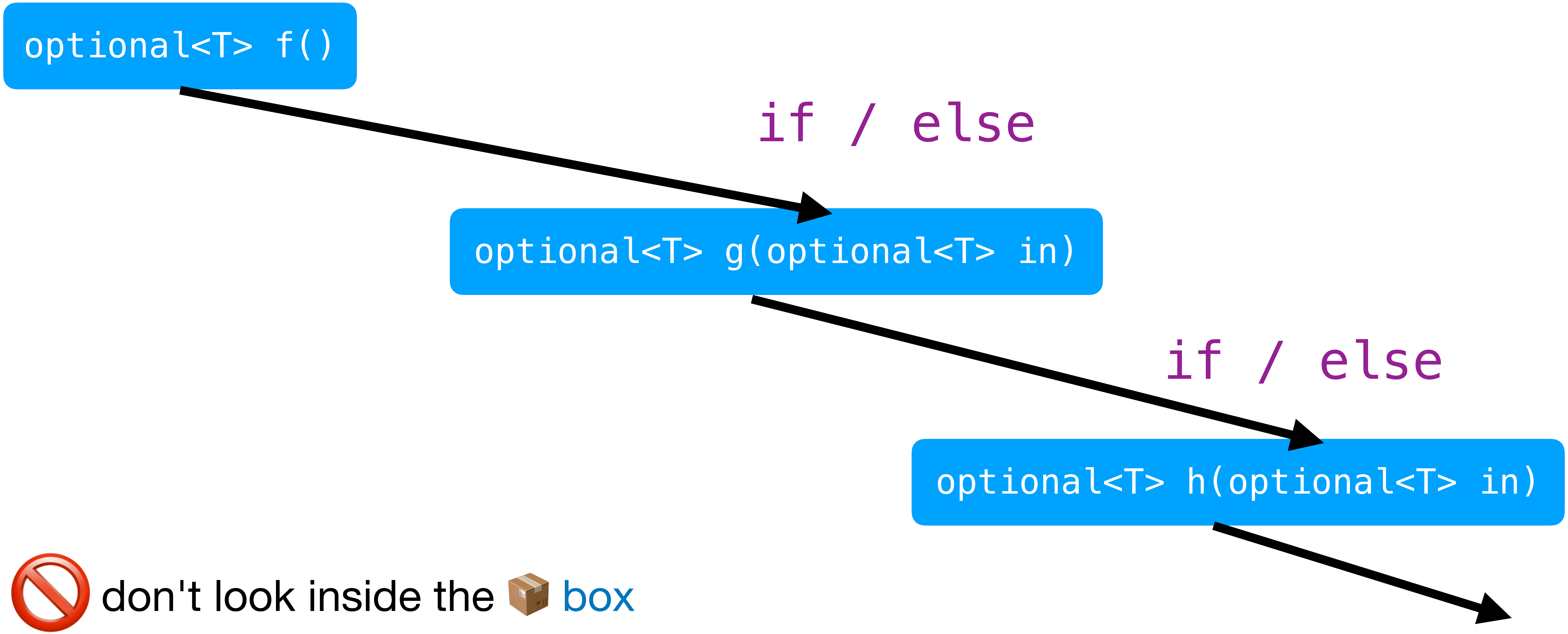
Why can't you give a Rustacian a christmas present?

They unwrap everything right away.

1:26 PM · Nov 14, 2022 · TweetDeck

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

📦 The Box



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

if / else

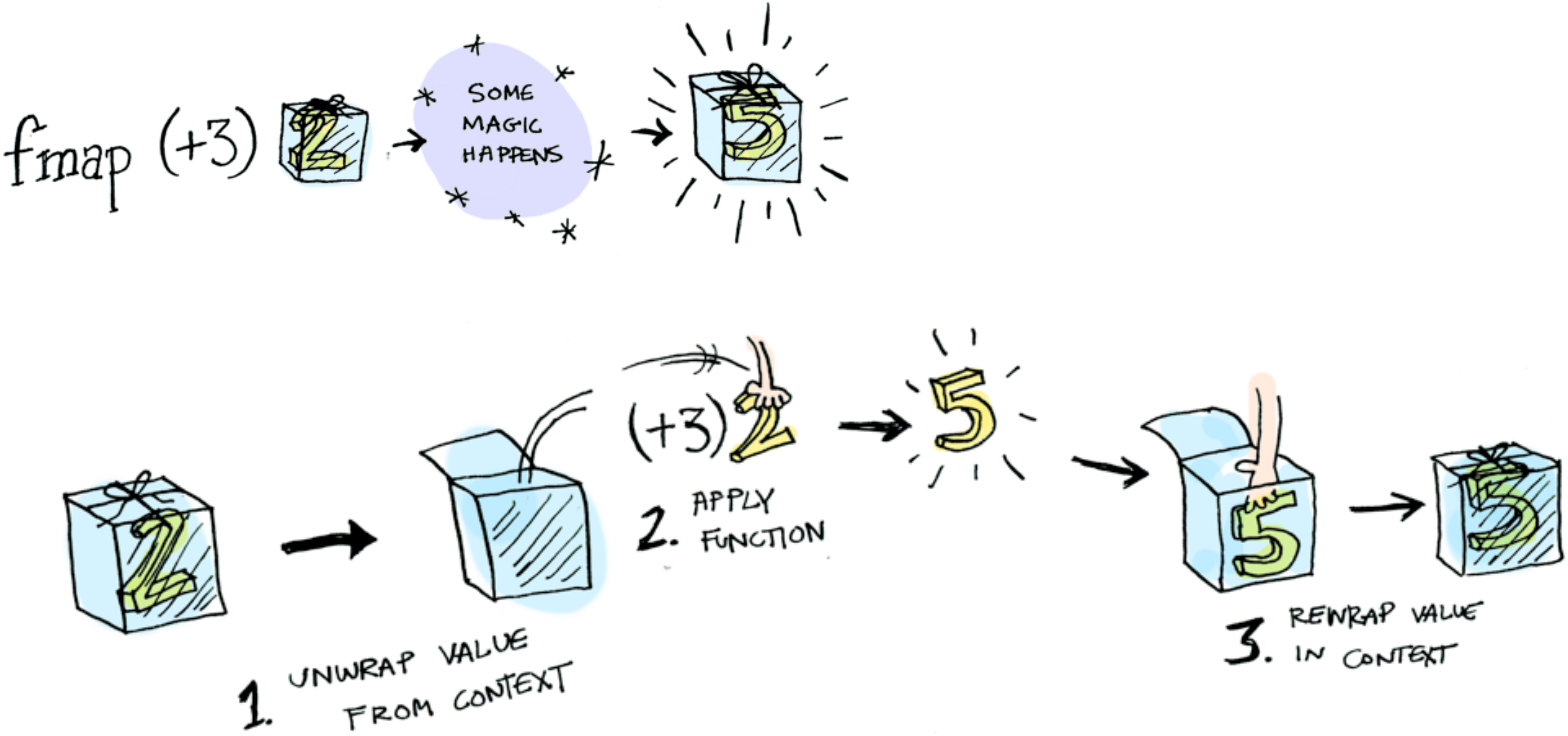
```
optional<T> g(optional<T> in)
```

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

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

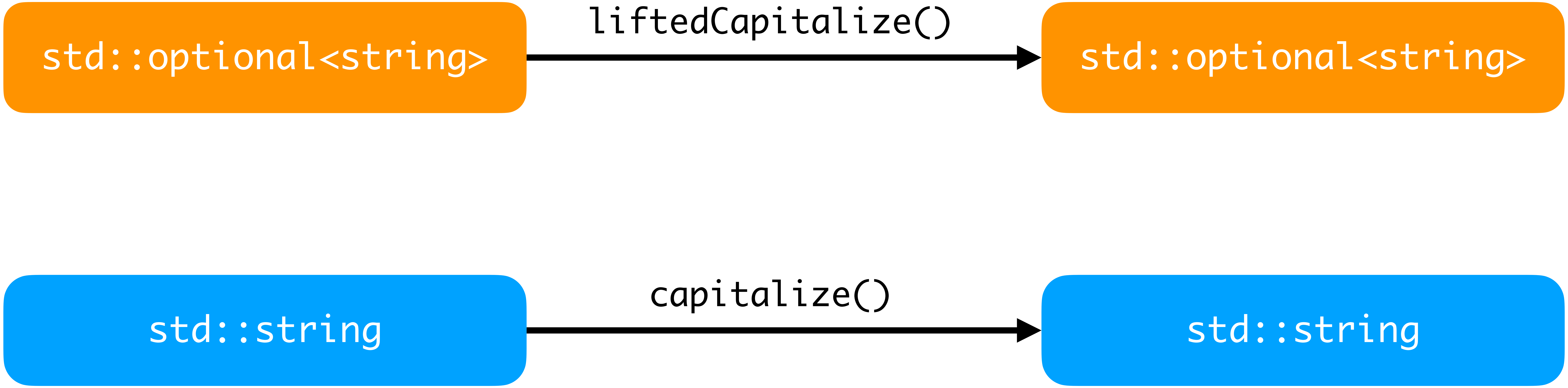
Lifting capitalize()

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()



std::optional<string>

liftedCapitalize()

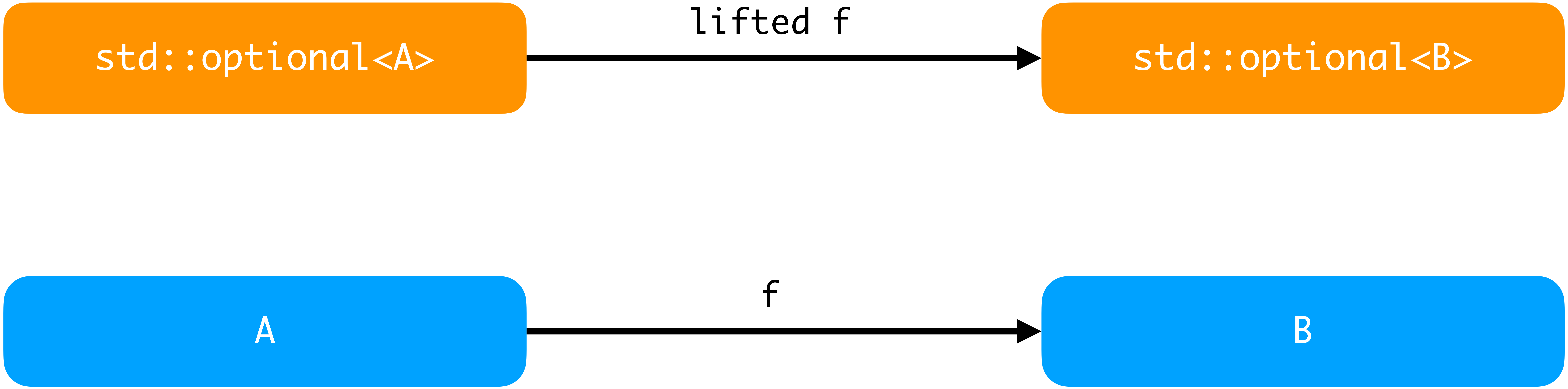
std::optional<string>

std::string

capitalize()

std::string

Lifting any function



std::optional<A>

lifted f

std::optional<B>

A

f

B

Lifting any function

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

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

  return result;
}
```

Lifting any function (take 2)

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

Lifting a function to a vector

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

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

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

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

Composition of lifted functions

The real power of lifted functions shines when composing functions.

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

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

liftedTrim

liftedLength

std::optional<string>

std::optional<string>

std::optional<int>

trim

length

std::string

std::string

int

📦 The Box

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

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

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

📦 The Box

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

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

Heritage

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


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


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

Heritage

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


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


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

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

Heritage / Concepts

`transform()`	functor	`fmap`
`and_then()`	monad	`>>= (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)
```

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)
```

Availability

std::optional

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

C++ 17

Availability

std::expected

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

C++ 23

Availability

.then()

Monadic operations for
std::optional (P0798)

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

C++ 23

Monadic operations for
std::expected (P2505)

- libstdc++ GCC 13
- libc++ Clang N/A
- Microsoft STL VS2022 17.6

Availability

Are we there yet?

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

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

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

Read more

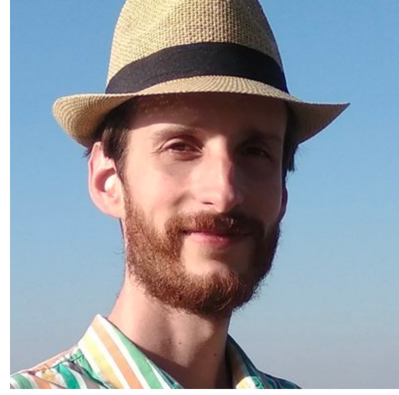
Sy Brand

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

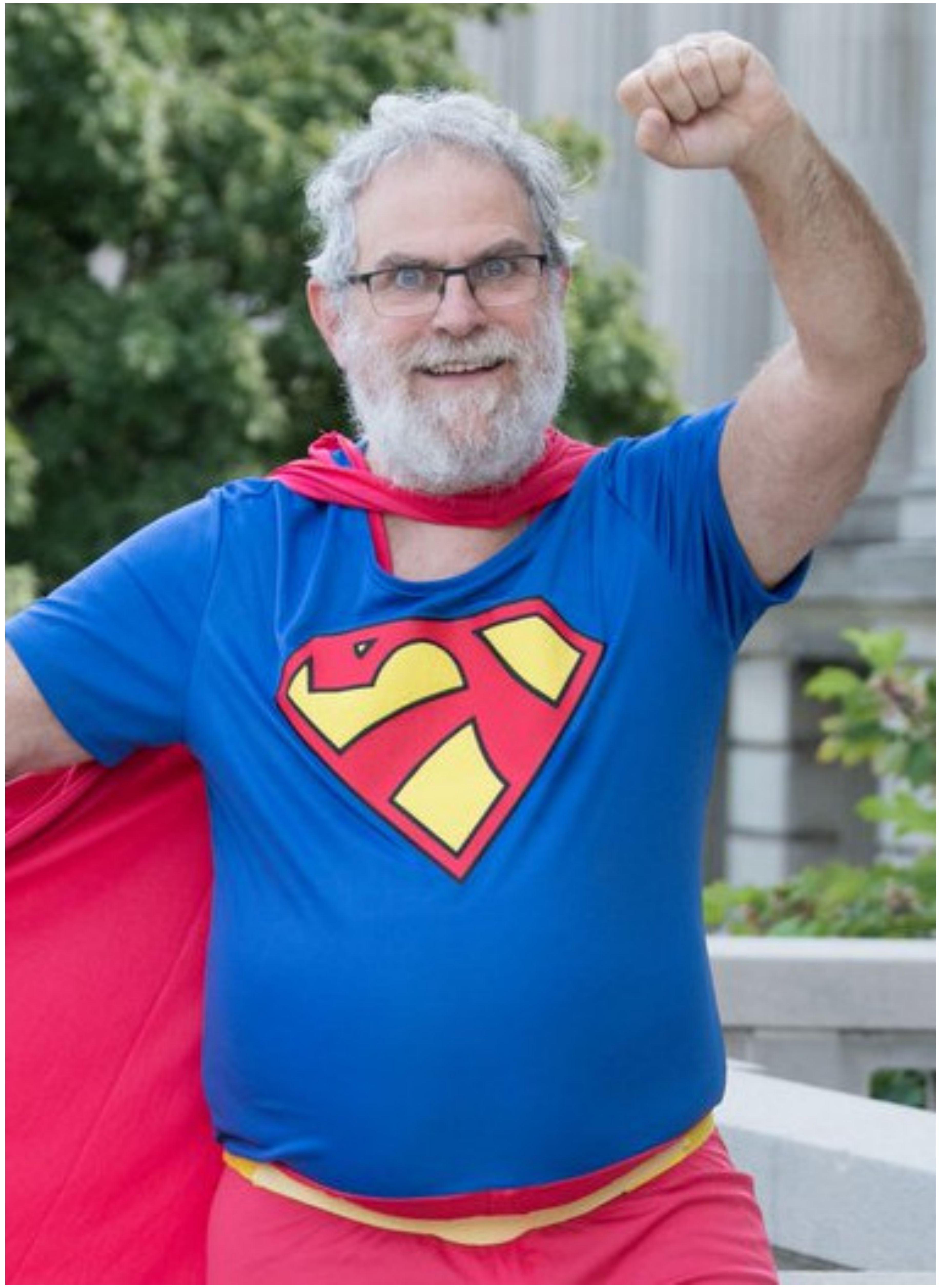
https://devblogs.microsoft.com/cppblog/cpp23s-optional-and-expected/

1990s, before monads...

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

`.and_then()` I'm done 😄

And Then() Some(T)

ACCU

April 2023

🐦 @ciura_victor
🐘 @ciura_victor@hachyderm.io

Victor Ciura
Principal Engineer
Visual C++