Monads in Modern C++

GEORGI KOYRUSHKI
& ALISTAIR FISHER

2023

October 01 - 06

Monads in Modern C++

Bloomberg

Engineering

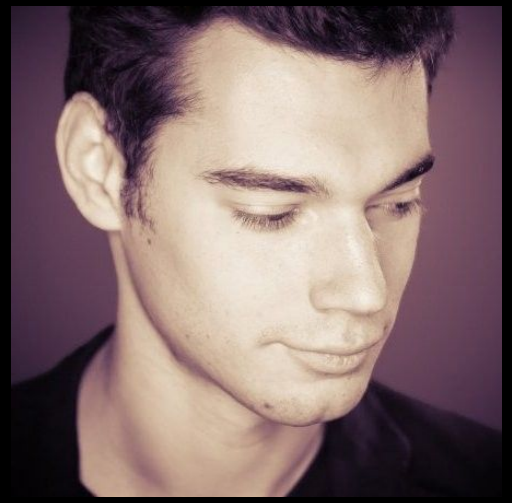
ACCU 2023
April 20, 2023

Georgi Koyrushki, Senior Software Engineer
Alistair Fisher, Engineering Team Lead

TechAtBloomberg.com

Speaker introductions

Georgi Koyrushki
Senior Software Engineer,
Multi-Asset Risk System (MARS)



Alistair Fisher
Engineering Team Lead, MARS

TechAtBloomberg.com

Bloomberg

Engineering

Agenda

- Motivation: Why do we care about monads?
- Theoretical basis: What are monads?
- Beyond theory: Monads in the STL and the wider world

What is a monad?

“A monoid in the category of endofunctors”

The curse of the monad

“In addition to its being good and useful, it’s also cursed and the curse of the monad is that once you get the epiphany, once you understand - "oh that's what it is" - you lose the ability to explain it to anybody else.”

- Douglas Crockford

TechAtBloomberg.com

Bloomberg

Engineering

Monad: a definition

In functional programming, a monad is a structure that combines program fragments (functions) and wraps their return values in a type with additional computation. In addition to defining a wrapping monadic type, monads define two operators: one to wrap a value in the monad type, and another to compose together functions that output values of the monad type (these are known as monadic functions). General-purpose languages use monads to reduce boilerplate code needed for common operations (such as dealing with undefined values or fallible functions, or encapsulating bookkeeping code). Functional languages use monads to turn complicated sequences of functions into succinct pipelines that abstract away control flow, and side-effects.

Monad: a definition

In functional programming, a monad is a structure that combines program fragments (functions) and wraps their return values in a type with additional computation. In addition to defining a wrapping monadic type, monads define two operators: one to wrap a value in the monad type, and another to compose together functions that output values of the monad type (these are known as monadic functions). General-purpose languages use monads to reduce boilerplate code needed for common operations (such as dealing with undefined values or fallible functions, or encapsulating bookkeeping code). Functional languages use monads to turn complicated sequences of functions into succinct pipelines that abstract away control flow, and side-effects.

The concepts

What makes C++ qualified for this talk?

- C++ is very well known to be
  - Procedural
  - Imperative
  - OO
  - Generic
  - … AND…
  - Functional (as of C++11)
- Functions in modern C++ are first-class citizens (ish)

What makes C++ qualified for this talk?

- C++ is very well known to be
  - Procedural
  - Imperative
  - OO
  - Generic
  - … AND…
  - Functional (as of C++11)
- Functions in modern C++ are first-class citizens (ish)

If `auto` is used as a type of a parameter or an explicit template parameter list is provided (since C++20), the lambda is a *generic lambda*. (since C++14)

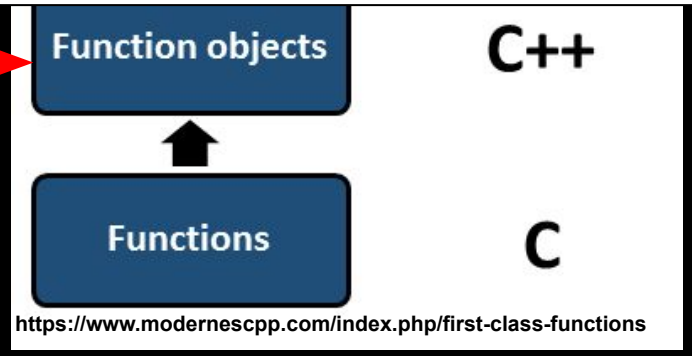
identifier initializer (3) (since C++14)

std::function

Defined in header <functional>

```
template< class >
class function; /* undefined */                    (since C++11)

template< class R, class... Args >
class function<R(Args...)>;                         (since C++11)
```

Overloading () operator on classes



https://www.modernescpp.com/index.php/first-class-functions

What is a Monad actually?

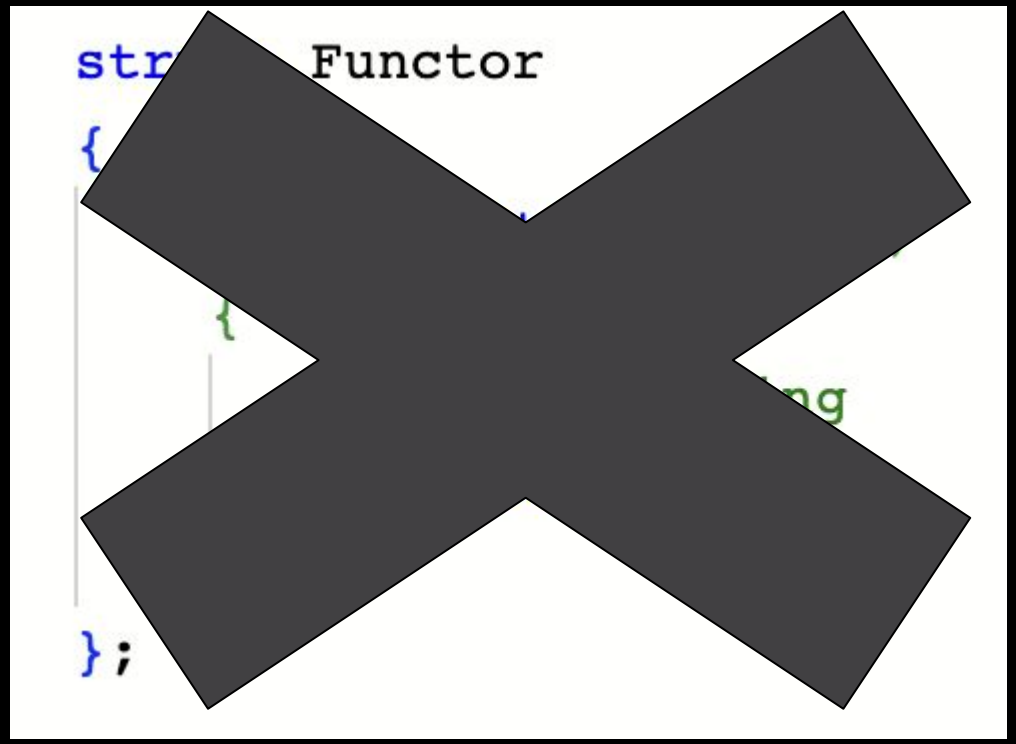
Let’s Start with Functors

TechAtBloomberg.com

Bloomberg

Engineering

What's a functor?



Let’s be (semi-) formal

Any type F that provides the following two operations:

- Given a type A, the ability to instantiate an instance of F<A> from an instance of A. In C++ terms: a constructor.
- The following operation, commonly called map, fmap or transform. Given an instance of F<A> and a function from A to B produce an instance of F<B>.
- A concept from category value theory

```
Functor<A>
    // aka map, fmap, etc.
    transform :: (A -> B) -> Functor<B>
```

C++ - IFIED

```
template<typename A>
struct Functor
{
    template <typename B>
    Functor<B> transform(std::function<B(A)>);
};
```

transform can also be defined externally

```
template <typename A, typename B>
Functor<B> transform(Functor<A>, std::function<B(A)>);
```

Example: Our First Functor - std::vector

```
01: // somewhere in namespace std...
02:
03: template <typename A>
04: struct vector
05: {
06:     // ...
07:     template <typename B>
08:     vector<B> transform(function<B(A)> f)
09:     {
10:
11:
12:
13:
14:
15:
16:     }
17:     // ...
18: };
```

Example: Our First Functor - std::vector

```
01: // somewhere in namespace std...
02:
03: template <typename A>
04: struct vector
05: {
06:     // ...
07:     template <typename B>
08:     vector<B> transform(function<B(A)> f)
09:     {
10:         vector<B> out;
11:
12:
13:
14:
15:         return out;
16:     }
17:     // ...
18: };
```

Example: Our First Functor - std::vector

```
01: // somewhere in namespace std...
02:
03: template <typename A>
04: struct vector
05: {
06:     // ...
07:     template <typename B>
08:     vector<B> transform(function<B(A)> f)
09:     {
10:         vector<B> out;
11:         for (auto elA : *this)
12:         {
13:             out.emplace_back(f(elA));
14:         }
15:         return out;
16:     }
17:     // ...
18: };
```

Using our vector functor

- We have a vector of strings, representing numbers

  {“1”, “2”, “3”,....}
- We want to double the numbers and return them, again as strings

  {“2”, “4”, “6”,....}
- We have the following operations already:

```
int convertStringToInt(std::string);

int doubleInt(int);

std::string convertIntToString(int);
```

Without functor

```
01: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
02: {
03:     std::vector<std::string> output;
04:     for(auto str: input)
05:     {
06:         int asInt = convertStringToInt(str);
07:         int doubled = doubleInt(asInt);
08:         output.push_back(convertIntToString(doubled));
09:     }
10:     return output;
11: }
```

vs. with functor

```
01: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
02: {
03:     std::vector<std::string> output;
04:     for(auto str: input)
05:     {
06:         int asInt = convertStringToInt(str);
07:         int doubled = doubleInt(asInt);
08:         output.push_back(convertIntToString(doubled));
09:     }
10:     return output;
11: }
```

```
1: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
2: {
3:     return input
4:         .transform(convertStringToInt)
5:         .transform(doubleInt)
6:         .transform(convertIntToString);
7: }
```

vs. with functor

```
01: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
02: {
03:     std::vector<std::string> output;
04:     for(auto str: input)
05:     {
06:         int asInt = convertStringToInt(str);
07:         int doubled = doubleInt(asInt);
08:         output.push_back(convertIntToString(doubled));
09:     }
10:     return output;
11: }
```

```
1: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
2: {
3:     return input
4:         .transform(std::function{convertStringToInt})
5:         .transform(std::function{doubleInt})
6:         .transform(std::function{convertIntToString});
7: }
```

vs. with functor

```
01: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
02: {
03:     std::vector<std::string> output;
04:     for(auto str: input)
05:     {
06:         int asInt = convertStringToInt(str);
07:         int doubled = doubleInt(asInt);
08:         output.push_back(convertIntToString(doubled));
09:     }
10:     return output;
11: }
```

```
1: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
2: {
3:     return input
4:         .transform(convertStringToInt)
5:         .transform(doubleInt)
6:         .transform(convertIntToString);
7: }
```

vs. with functor

```
01: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
02: {
03:     std::vector<std::string> output;
04:     for(auto str: input)
05:     {
06:         int asInt = convertStringToInt(str);
07:         int doubled = doubleInt(asInt);
08:         output.push_back(convertIntToString(doubled));
09:     }
10:     return output;
11: }
```

```
1: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
2: {
3:
4:
5:
6:
7: }
```

```
1: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
2: {
3:     return input.transform(convertStringToInt).transform(
4:         doubleInt).transform(convertIntToString);
5: }
```

clang-format :(

Have we seen something like this elsewhere?

```
1: std::vector<std::string> doubleNumbers(std::vector<std::string> input)
2: {
3:     return input
4:         | ranges::views::transform(convertStringToInt)
5:         | ranges::views::transform(doubleInt)
6:         | ranges::views::transform(convertIntToString)
7:         | ranges::to<std::vector>;
8: }
```

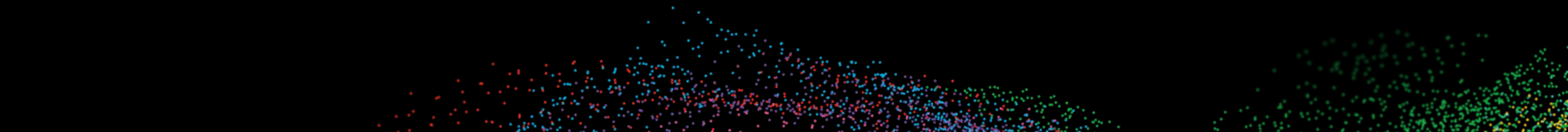
Why just std::vector?

```
1: std::optional<std::string> doubleNumbers(std::optional<std::string> input) {
2:     if (!input)
3:         return std::nullopt;
4:
5:     auto asInt = convertStringToInt(*input);
6:     auto doubled = doubleInt(asInt);
7:     auto toString = convertIntToString(doubled);
8:     return std::make_optional(toString);
9: }
```

```
1: std::optional<std::string> doubleNumbers(std::optional<std::string> input) {
2:     // similar to Haskell's Maybe::fmap
3:     return input
4:         .transform(convertStringToInt)
5:         .transform(doubleInt)
6:         .transform(convertIntToString);
7: }
```

std::optional functor

```
01: // somewhere in namespace std...
02:
03: template <typename A>
04: struct optional
05: {
06:     // ...
07:     template <typename B>
08:     optional<B> transform(function<B(A)> f)
09:     {
10:
11:
12:
13:
14:
15:
16:     }
17: };
```



std::optional functor

```
01: // somewhere in namespace std...
02:
03: template <typename A>
04: struct optional
05: {
06:     // ...
07:     template <typename B>
08:     optional<B> transform(function<B(A)> f)
09:     {
10:         optional<B> out;
11:
12:
13:
14:
15:         return out;
16:     }
17: };
```

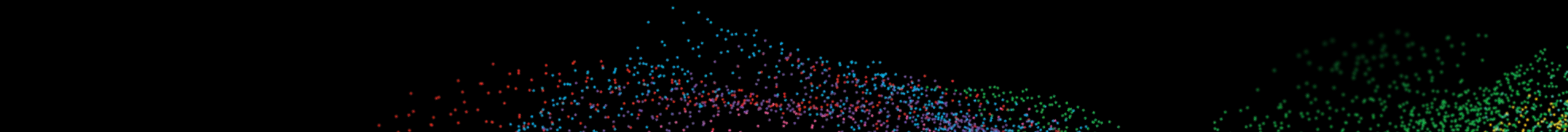
std::optional functor

```
01: // somewhere in namespace std...
02:
03: template <typename A>
04: struct optional
05: {
06:     // ...
07:     template <typename B>
08:     optional<B> transform(function<B(A)> f)
09:     {
10:         optional<B> out;
11:         if (this->has_value())
12:         {
13:             out = f(this->value())
14:         }
15:         return out;
16:     }
17: };
```

Why just std::vector?

```
1: std::optional<std::string> doubleNumbers(std::optional<std::string> input) {
2:     if (!input)
3:         return std::nullopt;
4:
5:     auto asInt = convertStringToInt(*input);
6:     auto doubled = doubleInt(asInt);
7:     auto toString = convertIntToString(doubled);
8:     return std::make_optional(toString);
9: }
```

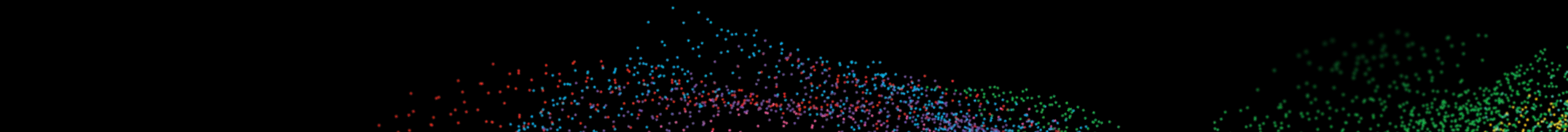
```
1: std::optional<std::string> doubleNumbers(std::optional<std::string> input) {
2:     // similar to Haskell's Maybe::fmap
3:     return input
4:         .transform(convertStringToInt)
5:         .transform(doubleInt)
6:         .transform(convertIntToString);
7: }
```

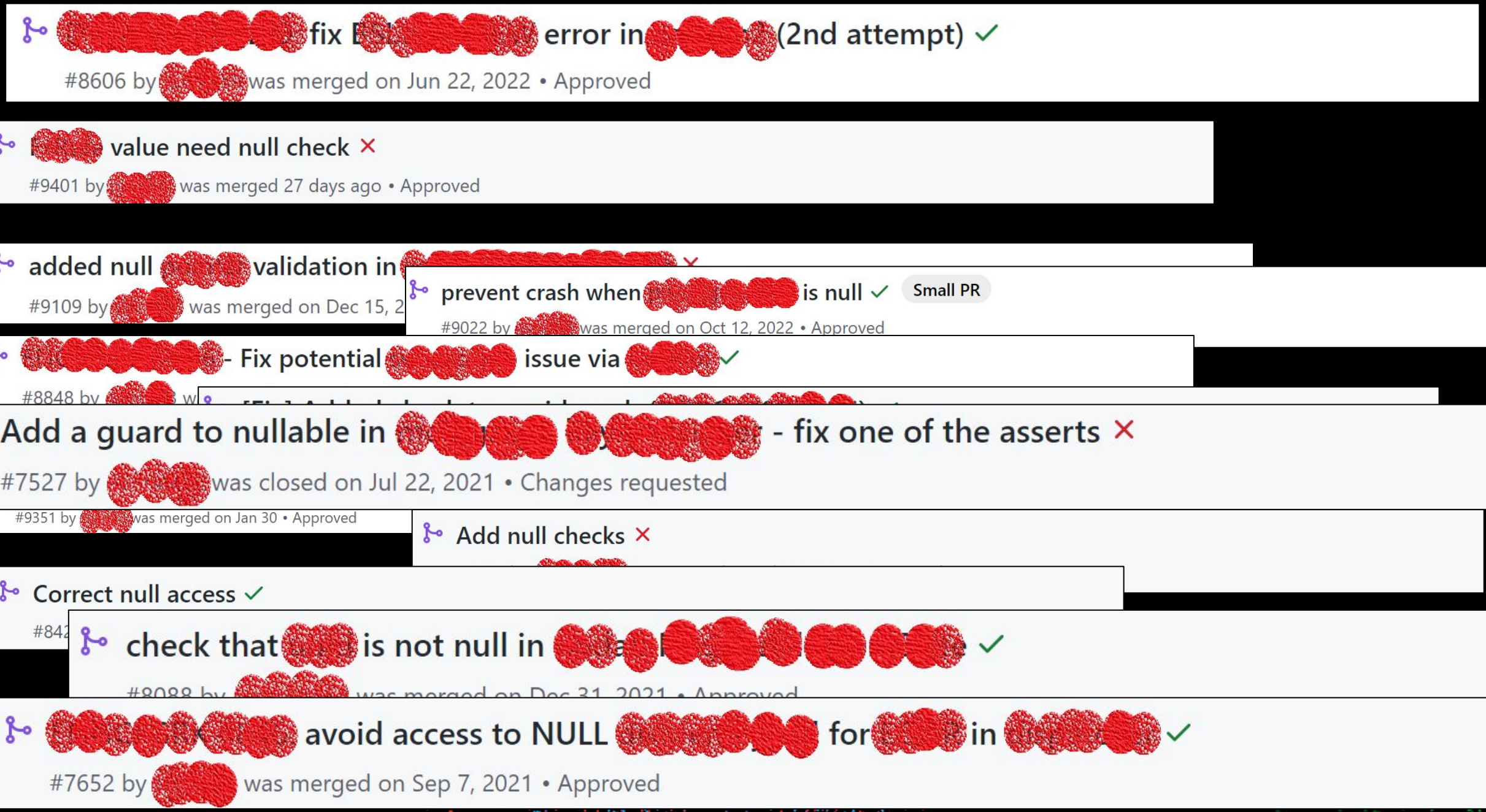


Why just std::vector?

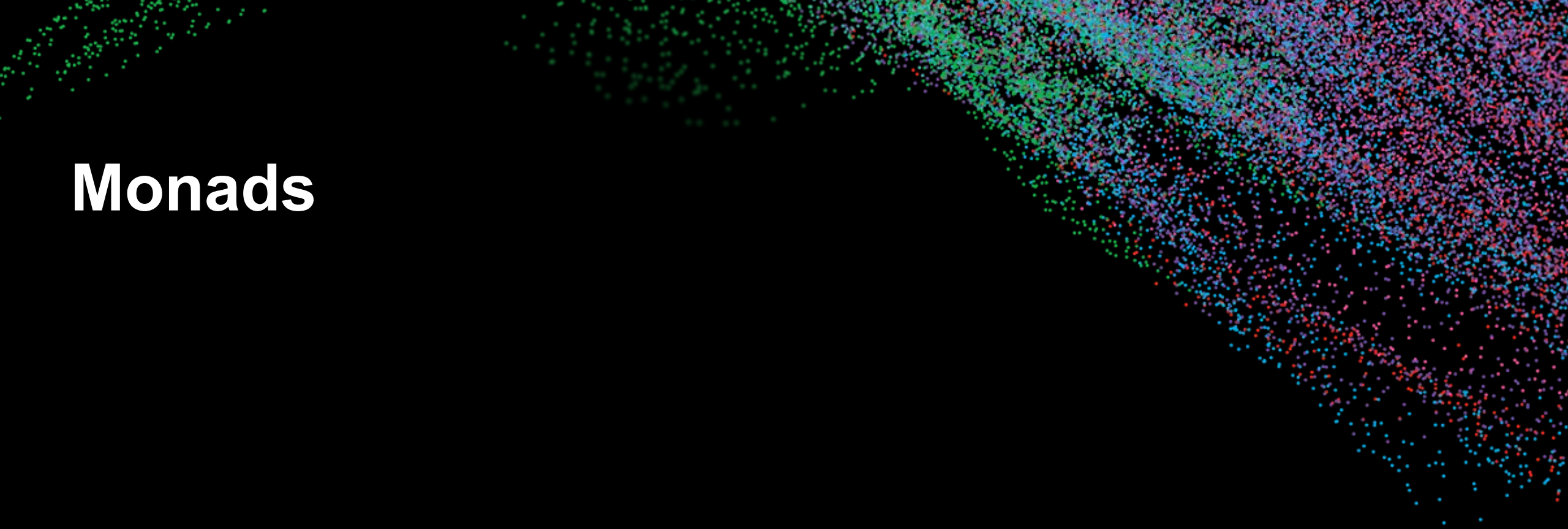
```
1: std::optional<std::string> doubleNumbers(std::optional<std::string> input) {
2:     if (!input)
3:         return std::nullopt;
4:
5:     auto asInt = convertStringToInt(*input);
6:     auto doubled = doubleInt(asInt);
7:     auto toString = convertIntToString(doubled);
8:     return std::make_optional(toString);
9: }
```

```
1: std::optional<std::string> doubleNumbers(std::optional<std::string> input) {
2:     // similar to Haskell's Maybe::fmap
3:     return input
4:         .transform(convertStringToInt) // if this.has_value() => convertStringToInt(*this)
5:         .transform(doubleInt) // if this.has_value() => doubleInt(*this) ...
6:         .transform(convertIntToString);
7: }
```




fix error in (2nd attempt)
#8606 by was merged on Jun 22, 2022 • Approved
value need null check
#9401 by was merged 27 days ago • Approved
added null validation in
#9109 by was merged on Dec 15, 2
prevent crash when is null
Small PR
#9022 by was merged on Oct 12, 2022 • Approved
- Fix potential issue via
#8848 by
Add a guard to nullable in - fix one of the asserts
#7527 by was closed on Jul 22, 2021 • Changes requested
#9351 by was merged on Jan 30 • Approved
Add null checks
Correct null access
#842
check that is not null in
#8088 by
avoid access to NULL for in
#7652 by was merged on Sep 7, 2021 • Approved

Monads



TechAtBloomberg.com

Bloomberg

Engineering

Where the functor starts to break down

```
int convertStringToInt(const string&);

int doubleInt(int i);

std::string convertIntToString(int i);
```

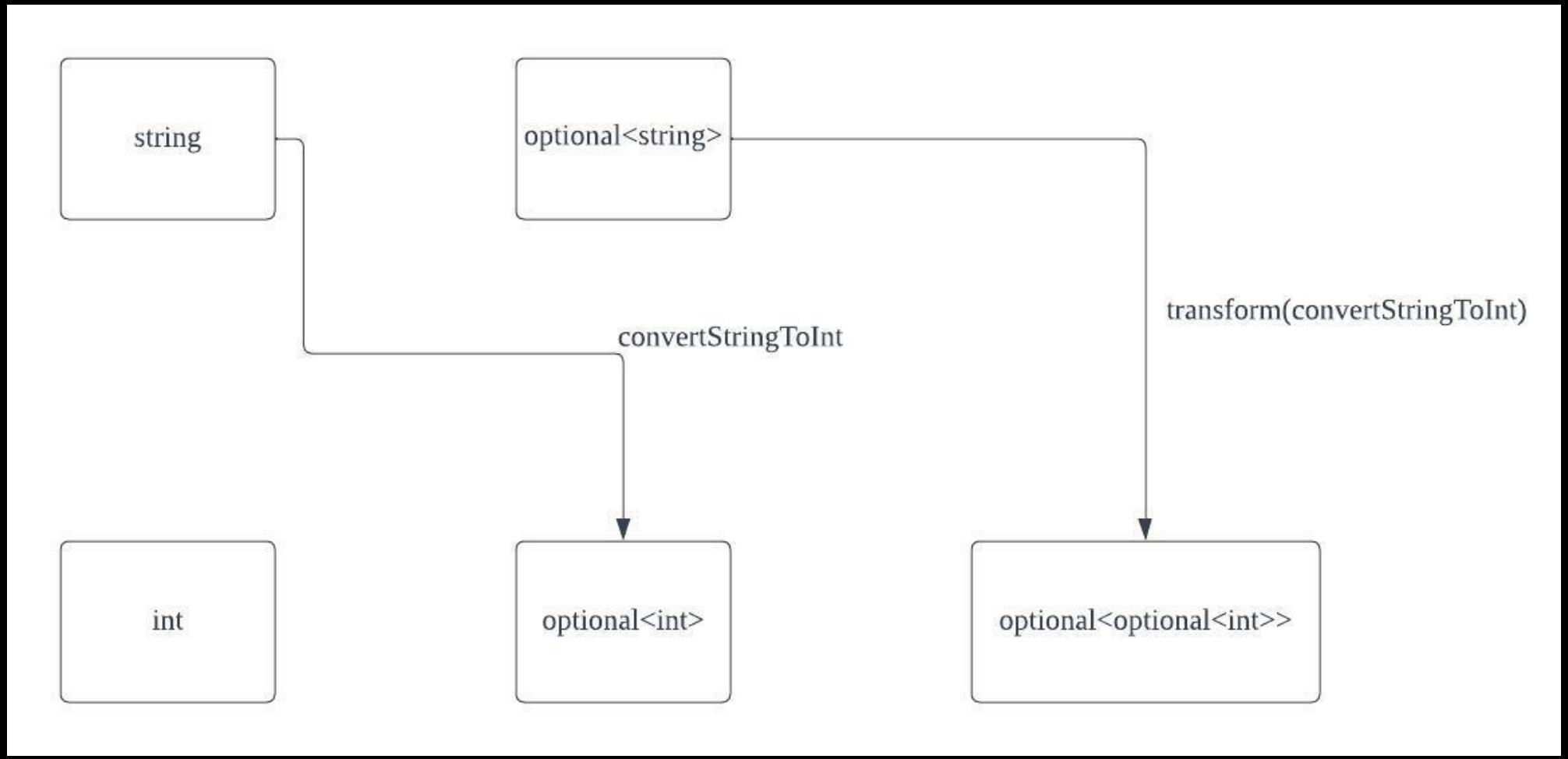
```
std::optional<int> convertStringToInt(string);

int doubleInt(int i);

string convertIntToString(int i);
```

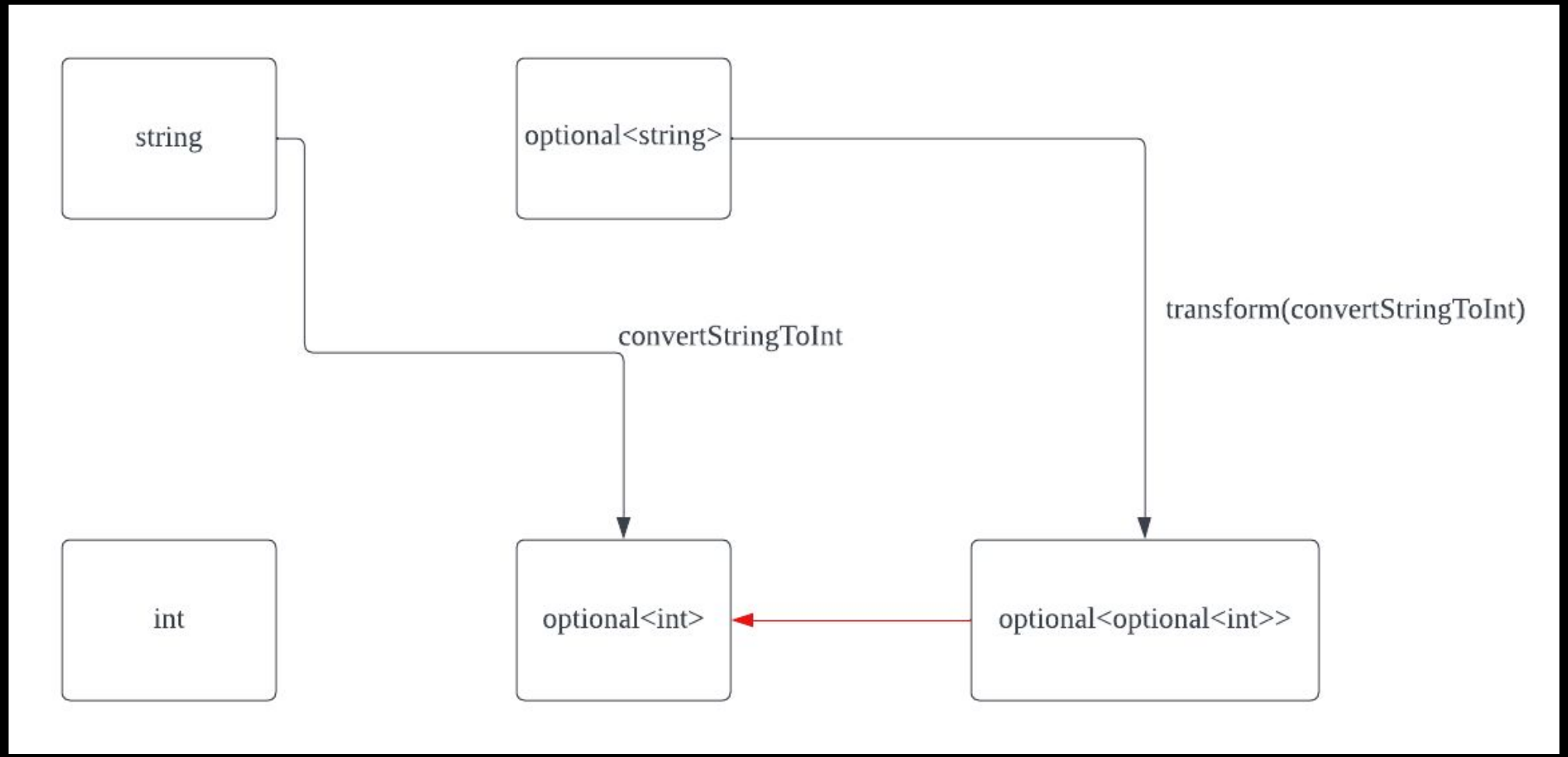
Oh

```
In file included from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/move.h:57,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/bits/stl_function.h:60,
                 from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/functional:49,
                 from <source>:1:
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/type_traits: In substitution of 'template<class _Fn, class ... _Args> using invoke_result_t = typename std::invoke_result::type [with _Fn = int (&)(in
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/optional:1125:10:   required from 'constexpr auto std::optional<_Tp>::transform(_Fn&&) && [with _Fn = int (&)(int); _Tp = std::optional<int>]'
<source>:113:23:   required from here
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/type_traits:3034:11: error: no type named 'type' in 'struct std::invoke_result<int (&)(int), std::optional<int> >'
 3034 |     using invoke_result_t = typename invoke_result<_Fn, _Args...>::type;
      |           ^~~~~~~~~~~~~~~
In file included from /opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/ranges:44,
                 from <source>:4:
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/optional: In instantiation of 'constexpr auto std::optional<_Tp>::transform(_Fn&&) && [with _Fn = int (&)(int); _Tp = std::optional<int>]':
<source>:113:23:   required from here
/opt/compiler-explorer/gcc-12.2.0/include/c++/12.2.0/optional:1127:20: error: no type named 'type' in 'struct std::invoke_result<int (&)(int), std::optional<int> >'
 1127 |             return optional<_Up>(_Optional_func<_Fn>{__f}, std::move(**this));
      |                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

What happened?

```
01: optional<string> doubleNumberFunctor(const optional<string>& opt)
02: {
03:   return opt.transform(convertStringToInt)   // optional<optional<int>>
04:             .transform(doubleInt)            // compile error - can't apply
05:                                              // doubleInt to optional<int>
06:             .transform(convertIntToString);
07:}
```





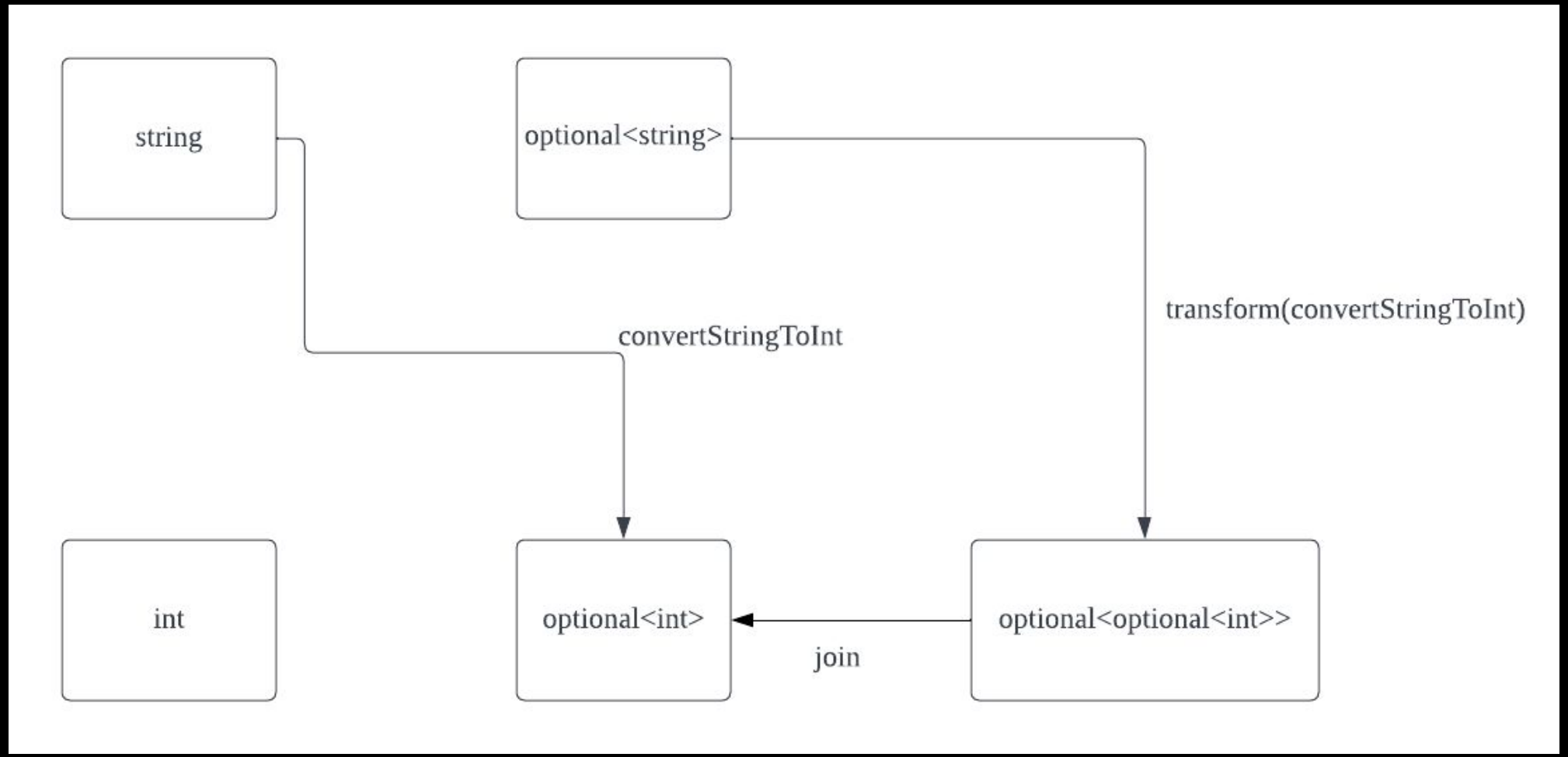
TechAtBloomberg.com

Bloomberg

Engineering

Join

```
01: template<typename T>
02: optional<T> join(const optional<optional<T>>& opt)
03: {
04:   if(opt.has_value())
05:   {
06:      return opt.value();
07:   }
08:   return {};
09:}
```



TechAtBloomberg.com

Bloomberg

Engineering

Fixing the example

```
01: optional<string> doubleNumberFunctor(const optional<string>& opt)
02: {
03:   auto nestOpt = transform(convertStringToInt) // optional<optional<int>>
04:
05:   return join(nestOpt)                        // optional<int>
06:             .transform(doubleInt)             // optional<int>
07:  .           .transform(convertIntToString); // optional<string>
08:}
```

Let’s introduce a new member function

```
01: template<typename A>
02: class optional {
03: // other functions...
04:
05: template <typename B>
06: optional<B> and_then(function<optional<B>(A)> f)
07: {
08:     auto optionalOptional = this -> transform(f); // have optional<optional<B>>
09:     return join(optionalOptional);
10:}
11:}
```

A nice pipeline again

```
01: optional<string> doubleNumberMonad(const optional<string>& opt)
02: {
03:   return opt.and_then(convertStringToInt)    // optional<int>
04:             .transform(doubleInt)            // optional<int>
05:   .         .transform(convertIntToString); // optional<string>
06:}
```

TechAtBloomberg.com

Bloomberg

Engineering

Introducing the monad

- All the requirements of functor, plus one more operation
- The operation is typically called monadic bind or mbind.
  In C++: and_then

```
template<typename A, typename B>
Monad<B> and_then(Monad<A>, function<Monad<B>(A)>);
```

- Can be thought of as a *transform*, followed by a *join*

Comparison to Functor

```
template<typename A, typename B>
Functor<B> transform(Functor<A>, function<B(A)>);

template<typename A, typename B>
Monad<B> and_then(Monad<A>, function<Monad<B>(A)>);
```

Another example: Vector

- Functor/transform: 1 to 1. Input vector size == output vector size
- Monad: 1 to any number. Output vector size can be anything

A new example - A C++ compiler

- We have a function from filename to compile errors

```
vector<string> compileFile(const string& filename);
```

- How can we turn a vector of files into a vector of compilation errors?

Start with transform…

```
01: vector<string> compileFiles(const vector<string>& filenames);
02: {
03: return filenames
04:         | ranges::views::transform(compileFile) // have vector<vector<string>...
05:                                                 // this feels familiar...
06:
07: };
```

Add a join…

```
01: vector<string> compileFiles(const vector<string>& filenames);
02: {
03: return filenames
04:         | ranges::views::transform(compileFile)
05:         | ranges::views::join
06:         | ranges::to<vector>
07: };
```

TechAtBloomberg.com

Bloomberg

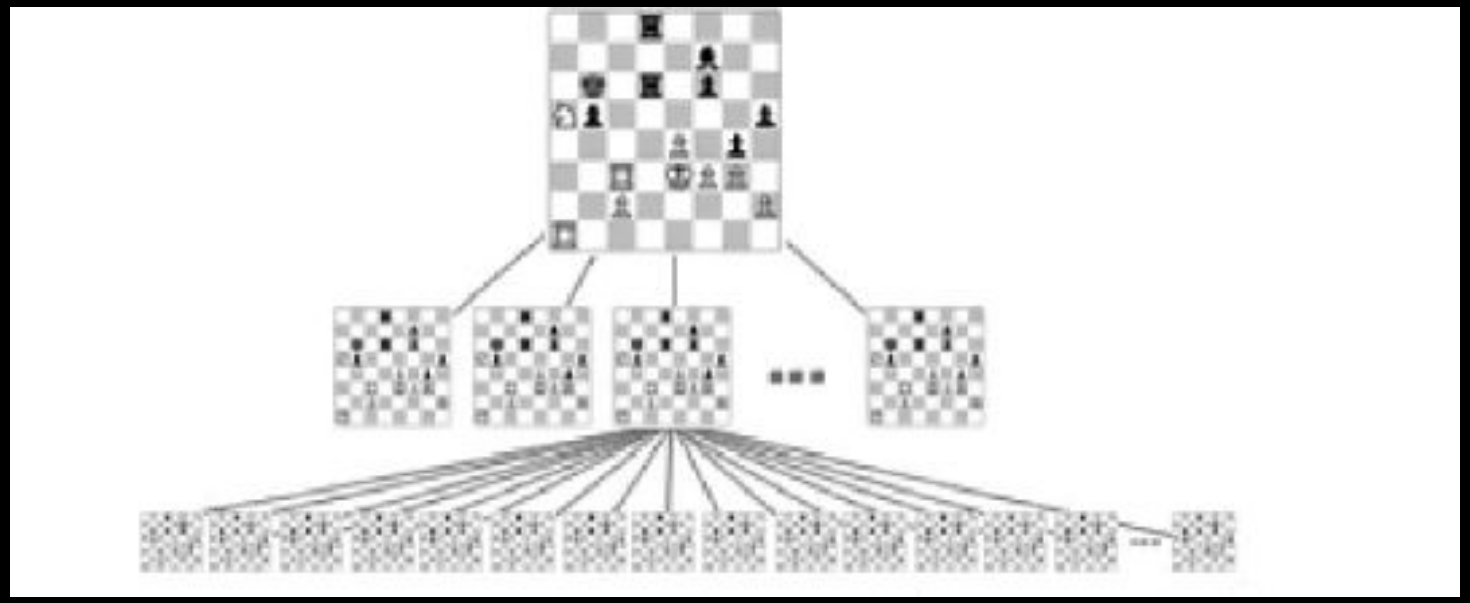
Engineering

Other use cases for vector monad

- Filtering - represent your predicate as element -> vector (empty is false, non empty is true)
- Simulations

Simulation

- World with a state and a function to create new states
- e.g., chess



Quick code example

```
01: struct ChessState
02: {
03: enum class Player{Black, White};
04: enum class GameState{WhiteWin, BlackWin, Stalemate, Ongoing};
05:
06: BoardState boardState; // some representation of the board state
07: GameState gameState;
08: Player currentPlayer;
09:
10: ChessState(){...} // set up correct starting position
11: };
```

```
vector<ChessState> evolveBoard(const ChessState& state);
```

A quick chess simulator

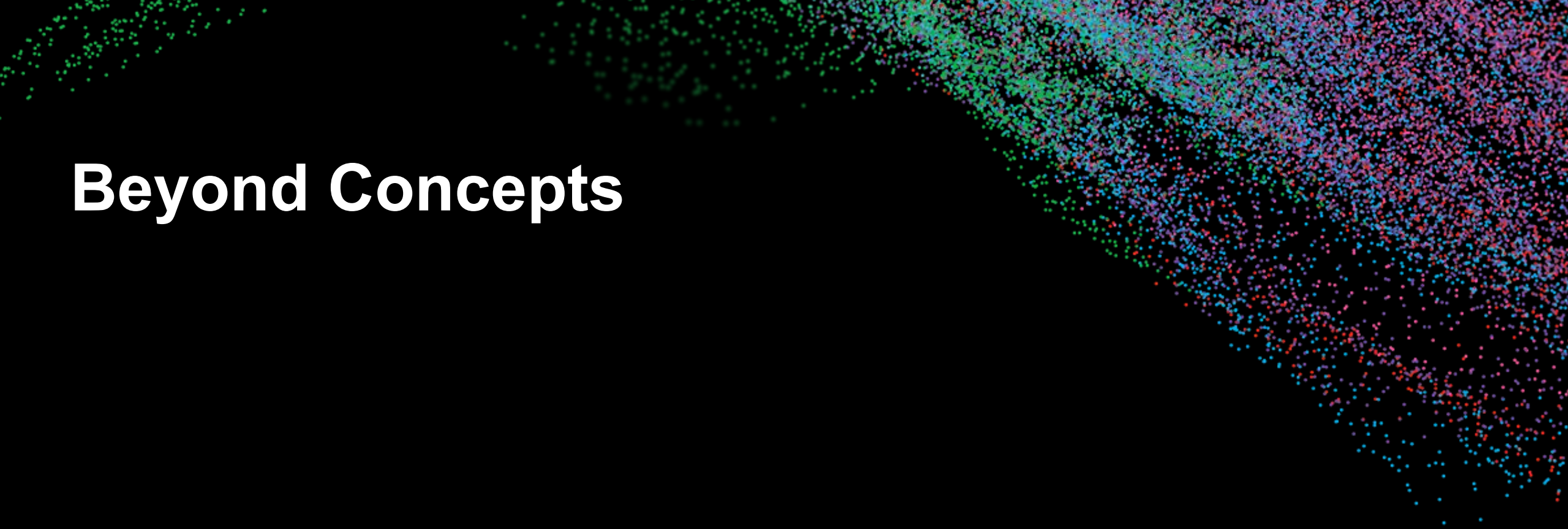
```
01: vector<ChessState> evolveBoards(const vector<ChessState>& states);
02: {
03: return states
04:         | ranges::views::transform(evolveBoard)
05:         | ranges::views::join
06:         | ranges::to<vector>
07: };
```

```
01: // Generate 50 turns
02:
03: vector<ChessState> states{initialBoardState};
04:
05: for(int i = 0; i < 50; ++i)
06:     states = evolveBoards(states);
```

Common monad types

- Maybe monad (optional monad)
- List monad (vector monad)
- IO Monad
- Reader Monad
- Writer Monad
- State Monad

Beyond Concepts



TechAtBloomberg.com

Bloomberg

Engineering

What’s available?

std::optional

```cpp
#include <optional>
#include <vector>
#include <iostream>
#include <string>
#include <charconv>
#include <ranges>
std::optional<int> to_int(std::string_view sv)
{
  int r{};
  auto [ptr, ec] { std::from_chars(sv.data(), sv.data() + sv.size(), r) };
  if (ec == std::errc())
    return r;
  else
    return std::nullopt;
}

int main()
{
    using namespace std::literals;

    std::vector<std::optional<std::string>> v = {
     "1234", "15 foo", "bar", "42", "5000000000", " 5"
    };

    for( auto&& x : v | std::views::transform([](auto&& o) { return
          o.and_then(to_int)                    // flatmap from strings to ints
                                                // (making empty optionals where it fails)
           .transform([](int n) { return n+1; })          // map int to int + 1
           .transform([](int n) { return std::to_string(n); }) // convert back to strings
           .or_else([]{ return std::optional{"Null"s}; }); // replace all empty optionals
                                                // that were left by and_then and
                                                // ignored by transforms with "Null"
       }); }) )
      std::cout << *x << '\n';
}
```

tl::optional

```
int i
tl::op
*o ==
i = 12
*o ==
&*o ==
```

```
auto f7
auto o7
STATIC_
    (st
REQUIRE
```

The interface is the same as `std::optional`, but the following member functions are also defined. Explicit types are for clarity.

- `map` : carries out some operation on the stored object if there is one.
  - `tl::optional<std::size_t> s = opt_string.map(&std::string::size);`
- `and_then` : like `map`, but for operations which return a `tl::optional`.
  - `tl::optional<int> stoi (const std::string& s);`
  - `tl::optional<int> i = opt_string.and_then(stoi);`
- `or_else` : calls some function if there is no value stored.
  - `opt.or_else([] { throw std::runtime_error{"oh no"}; });`
- `map_or` : carries out a `map` if there is a value, otherwise returns a default value.
  - `tl::optional<std::size_t> s = opt_string.map_or(&std::string::size, 0);`
- `map_or_else` : carries out a `map` if there is a value, otherwise returns the result of a given default function.
  - `std::size_t get_default();`
  - `tl::optional<std::size_t> s = opt_string.map_or_else(&std::string::size, get_default);`
- `conjunction` : returns the argument if a value is stored in the optional, otherwise an empty optional.
  - `tl::make_optional(42).conjunction(13); //13`
  - `tl::optional<int>{}.conjunction(13); //empty`
- `disjunction` : returns the argument if the optional is empty, otherwise the current value.
  - `tl::make_optional(42).disjunction(13); //42`
  - `tl::optional<int>{}.disjunction(13); //13`
- `take` : returns the current value, leaving the optional empty.
  - `opt_string.take().map(&std::string::size); //opt_string now empty;`

std::expected

```cpp
#include <iostream>
#include <expected>
#include <stdexcept>
#include <string_view>

enum class parse_error {
    invalid_char,
    overflow,
    // ...
    unknown
};

std::expected<int, parse_error>
parse_number(const std::string& str) {
    try {
        return std::stoi(str);
    } catch (const std::invalid_argument &error) {
        return std::unexpected(parse_error::invalid_char);
    } catch (const std::out_of_range &error) {
        return std::unexpected(parse_error::overflow);
    } catch (...) {
        return std::unexpected(parse_error::unknown);
    }
}

int main() {
    auto res = parse_number("123")
        .transform([](auto val){ return val; })
        .transform_error([](auto error) { /* do error handling */ return error; })
        .or_else([] (auto /* error */) -> std::expected<int, parse_error>{ return -1; });
    return 0;
}
```

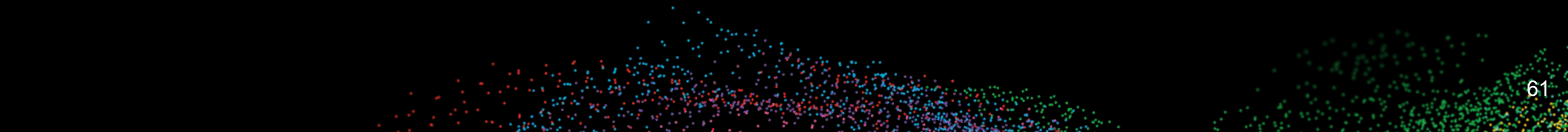
tl::expected

The interface is the same as `std::expected` as proposed in p0323r3, but the following member functions are also defined. Explicit types are for clarity.

- `map` : carries out some operation on the stored object if there is one.
  - `tl::expected<std::size_t,std::error_code> s = exp_string.map(&std::string::size);`
- `map_error` : carries out some operation on the unexpected object if there is one.
  - `my_error_code translate_error (std::error_code);`
  - `tl::expected<int,my_error_code> s = exp_int.map_error(translate_error);`
- `and_then` : like `map` , but for operations which return a `tl::expected` .
  - `tl::expected<ast, fail_reason> parse (const std::string& s);`
  - `tl::expected<ast, fail_reason> exp_ast = exp_string.and_then(parse);`
- `or_else` : calls some function if there is no value stored.
  - `exp.or_else([] { throw std::runtime_error{"oh no"}; });`

tl::optional and tl::expected by Sy Brand

Prior to C++23



std::ranges

C++ | Ranges library

Ranges library (C++20)

Range conversions

Defined in header <ranges>

ranges::to (C++23)	constructs a new non-view object from an input range (function template)

Range-v3

Range algorithms, views, and actions for the Standard Library

- Range-v3
  - User Manual
  - Examples
  - Release Notes
  - Modules
  - Namespaces
  - Concepts

User Manual

Preface

Range library for C++14/17/20. This code is the basis of the range support in C++20.

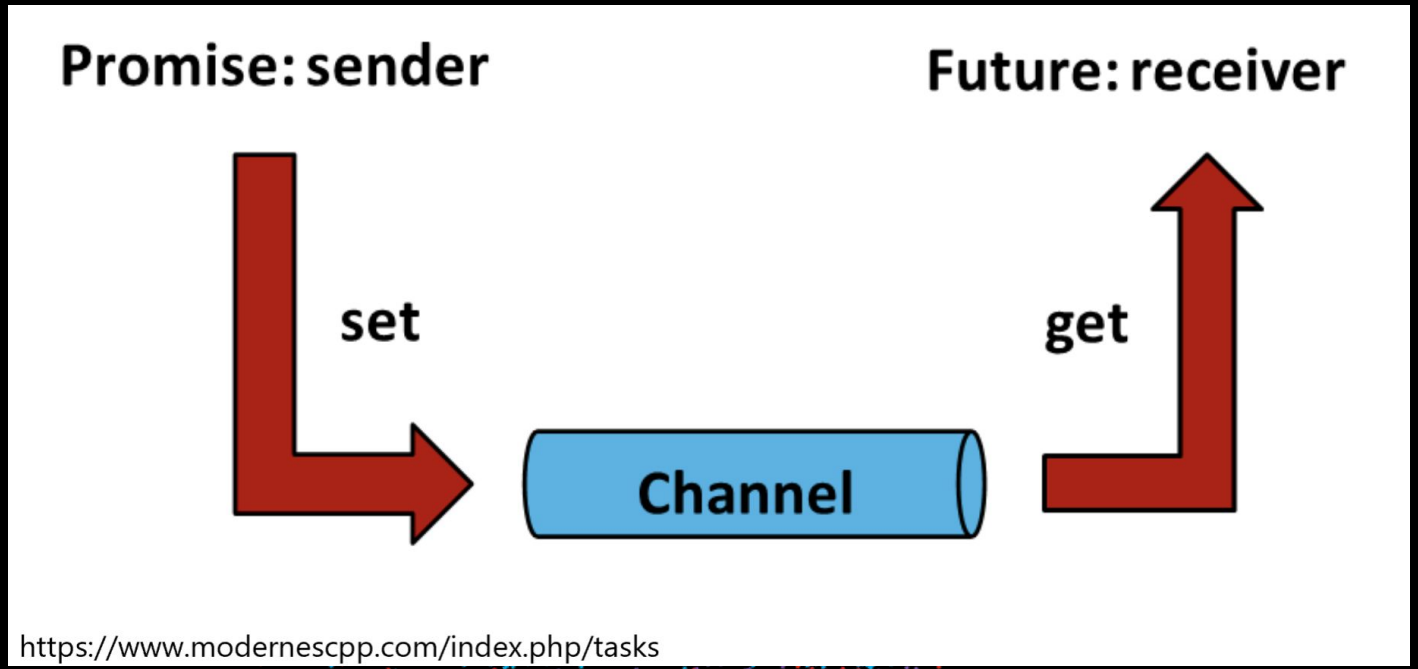
std::future

C++ | Concurrency support library | std::future

std::future

Defined in header <future>

```
template< class T > class future;          (1)  (since C++11)
template< class T > class future<T&>;      (2)  (since C++11)
template<>          class future<void>;    (3)  (since C++11)
```



https://www.modernescpp.com/index.php/tasks

std::future

```
01: // sender/producer
02: std::future<int> async_algo() {
03:     std::promise<int> p;
04:     auto f = p.get_future();
05:     std::thread t { [p = std::move(p)] () mutable {
06:         int answer = // compute!
07:         p.set_value(answer);
08:     }};
09:     t.detach();
10:     return f;
11: }
```

1. Blocking
2. No composability - aka continuation

```
1: // receiver/consumer
2: int main() {
3:     // get() blocks!
4:     auto res = async_algo().get();
5:     std::cout << "res = " << res << std::endl;
6: }
```

std::future

```
std::future<T1> f1 = ...;
```

```
T2 continuationA(T1);
auto f2 = f1.then(continuationA);
```

functor

```
std::future<T2> continuationB(T1);
auto f2 = f1.then(continuationB);
```

monad

```
auto res = f1
    .then(c1)
    .then(c2)
    // ...
    .then(cn).get();
```

std::future

```
std::future<T1> f1 = ...;
```

```
T2 continuationA(T1);
auto f2 = f1.then(continuationA);
```

functor

```
std::future<T2> continuationB(T1);
auto f2 = f1.then(continuationB);
```

monad

```
auto res = f1
    .then(c1)
    .then(c2)
    // ...
    .then(cn).get();
```

- Fix the std::future pattern
  - I See a Monad in Your Future (link)
  - Extensions for concurrency TS (link)
  - Unified Futures Proposal (p1054R0)
- Where does then execute?
- Rolled into std::execution framework (P2300R6)
  - Senders/Receivers
  - Generic async algorithms
  - Schedulers
- WIP :(

std::monad/std::functor

```
1: std::optional<std::string> doubleNumbers(
2:     const std::optional<std::string>& input)
3: {
4:     return std::functor::map(
5:             std::functor::map(
6:                 std::monad::bind(input, convertStringToInt),
7:                 doubleInt),
8:             convertIntToString);
9: }
```

- C++ Monadic Interface (p0650r0)
  - Proposal from 2017 (never made it :()
  - Generic type classes + customization points
- Unifies functor/monad in the standard
- Interesting questions raised
  - Are smart pointers (pointers) functors?
  - Are std::array, std::vector functors?
  - etc.
- Boost.Hana achieves this via Functor / Monad (and a few other) concepts
  - hana::lazy/hana::optional/hana::tuple

p1255: A view of 0 or 1 elements: views::maybe by Steve Downey

```
1: std::optional o{7}; // Also *s, smart ptrs, etc.
2: for (auto&& i : views::nullable(std::ref(o))) {
3:     i = 9;
4:     std::cout << "i=" << i << " prints 9\n";
5: }
6: std::cout << "o=" << *o << " prints 9\n";
```

- views::nullable
  - Extra safety for maybe-like types
  - std::optional, shared_ptr, raw ptrs, result of find (?)

How does this work?

```
01: template <typename Nullable>
02: class nullable_view<Nullable> : public ranges::view_interface<nullable_view<Nullable>> {
03: public:
04:     constexpr nullable_view() : value_(nullptr){};
05:
06:     constexpr explicit nullable_view(Nullable& nullable)
07:     : value_(std::addressof(nullable)) {}
08:     // ...more construction...
09:
10:     constexpr const T* data() const noexcept { /* implementation */ }
11:
12:     constexpr T*       begin() noexcept { return data(); }
13:     constexpr const T* begin() const noexcept { return data(); }
14:
15:     constexpr T*       end() noexcept { return data() + size(); }
16:     constexpr const T* end() const noexcept { return data() + size(); }
17:
18:     constexpr size_t size() const noexcept {
19:         if (!value_)
20:             return 0;
21:         return 1;
22:     }
23: private:
24:     Nullable* value_;
25: };
```

Adapter pattern

p1255: A view of 0 or 1 elements: views::maybe by Steve Downey

```
1: std::optional o{7}; // Also *s, smart ptrs, etc.
2: for (auto&& i : views::nullable(std::ref(o))) {
3:     i = 9;
4:     std::cout << "i=" << i << " prints 9\n";
5: }
6: std::cout << "o=" << *o << " prints 9\n";
```

- views::nullable
  - Extra safety for maybe-like types
  - std::optional, shared_ptr, raw ptrs, result of find (?)

- Give us the basis to implement anything!

p1255: A view of 0 or 1 elements: views::maybe by Steve Downey

```
01: std::optional<std::string> doubleNumbers(
02:     const std::optional<std::string>& input)
03: {
04:     auto doubled = std::views::nullable(std::ref(input))
05:         | std::views::transform(convertStringToInt)
06:         | std::views::transform(std::views::nullable)
07:         | std::views::join
08:         | std::views::transform(doubleInt)
09:         | std::views::transform(convertIntToString);
10:
11:      std::optional<std::string> out {};
12:      for (auto&& d : doubled)
13:          out.emplace(std::move(d));
14:      return out;
15: }
```

and_then

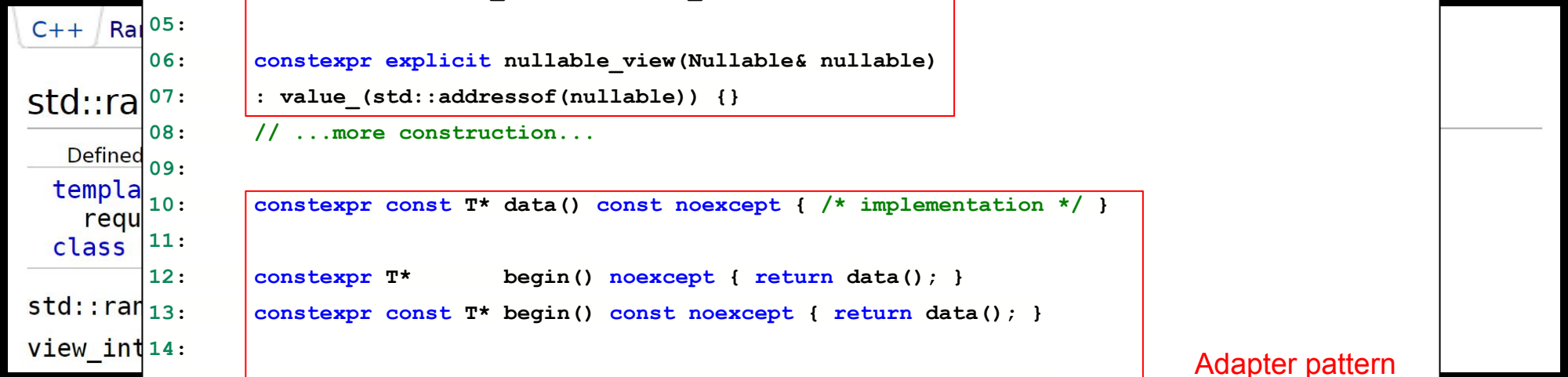
- views::nullable
  - Extra safety for maybe-like types
  - std::optional, shared_ptr, raw ptrs, result of find (?)
- Give us the basis to implement anything!

p1255: A view of 0 or 1 elements: views::maybe by Steve Downey

```
1: std::optional o{7}; // Also *s, smart ptrs, etc.
2: for (auto&& i : views::nullable(std::ref(o))) {
3:     i = 9;
4:     std::cout << "i=" << i << " prints 9\n";
5: }
6: std::cout << "o=" << *o << " prints 9\n";
```

- views::nullable
  - Extra safety for maybe-like types
  - std::optional, shared_ptr, raw ptrs, result of find (?)

How does this work?




```
01: template <typename Nullable>
02: class nullable_view<Nullable> : public ranges::view_interface<nullable_view<Nullable>> {
03: public:
04:     constexpr nullable_view() : value_(nullptr){};
05:
06:     constexpr explicit nullable_view(Nullable& nullable)
07:     : value_(std::addressof(nullable)) {}
08:     // ...more construction...
09:
10:     constexpr const T* data() const noexcept { /* implementation */ }
11:
12:     constexpr T*       begin() noexcept { return data(); }
13:     constexpr const T* begin() const noexcept { return data(); }
14:
15:     constexpr T*       end() noexcept { return data() + size(); }
16:     constexpr const T* end() const noexcept { return data() + size(); }
17:
18:     constexpr size_t size() const noexcept {
19:         if (!value_)
20:             return 0;
21:         return 1;
22:     }
23: private:
24:     Nullable* value_;
25: };
```

Adapter pattern

p1255: A view of 0 or 1 elements: views::maybe by Steve Downey

```
1: std::optional o{7}; // Also *s, smart ptrs, etc.
2: for (auto&& i : views::nullable(std::ref(o))) {
3:     i = 9;
4:     std::cout << "i=" << i << " prints 9\n";
5: }
6: std::cout << "o=" << *o << " prints 9\n";
```

```
1: maybe_view<int> mv{10};
2: for (auto&& i : mv) {
3:     std::cout << "i=" << i << " prints 10\n";
4: }
```

- views::nullable
  - Extra safety for maybe-like types
  - std::optional, shared_ptr, raw ptrs, result of find (?)
- maybe_view
  - Part range (can iterate over, etc.)
  - Part optional (at most 1 value)
- Give us the basis to implement anything!

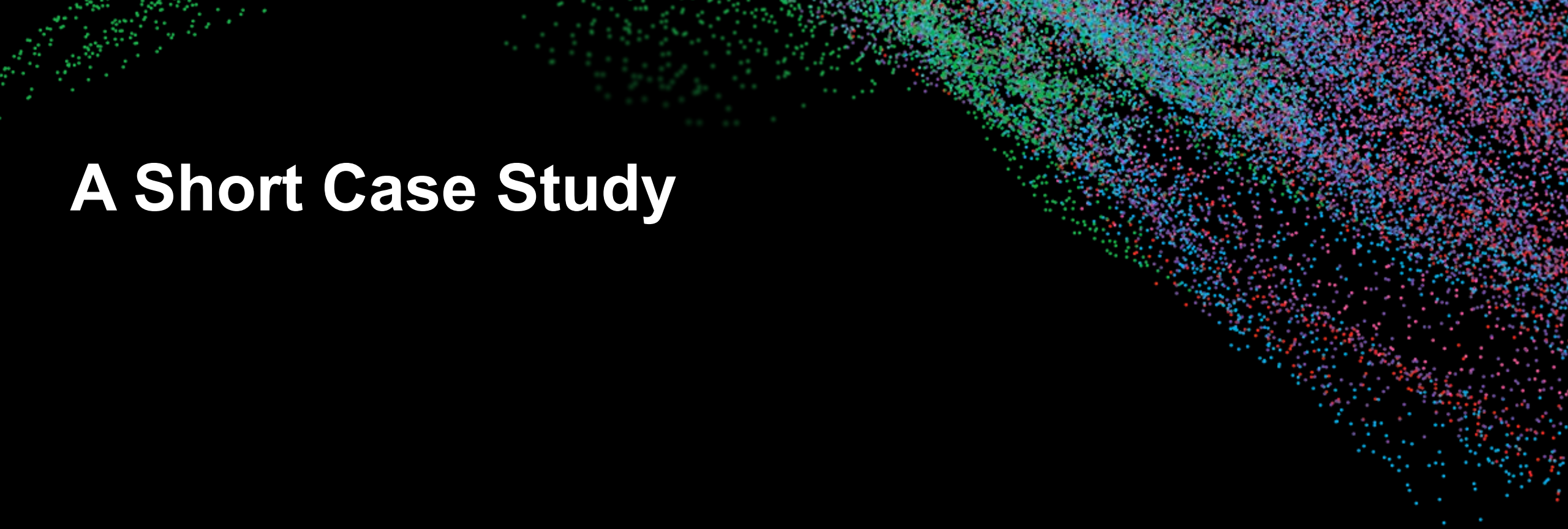
p1255: A view of 0 or 1 elements: views::maybe by Steve Downey

```
01: std::optional<std::string> doubleNumbers(
02:     const std::optional<std::string>& input)
03: {
04:     auto doubled = std::views::nullable(std::ref(input))
05:         | std::views::transform(convertStringToInt)
06:         | std::views::transform(std::views::nullable)
07:         | std::views::join
08:         | std::views::transform(doubleInt)
09:         | std::views::transform(convertIntToString);
10:
11:      std::optional<std::string> out {};
12:      for (auto&& d : doubled)
13:          out.emplace(std::move(d));
14:      return out;
15: }
```

and_then

- views::nullable
  - Extra safety for maybe-like types
  - std::optional, shared_ptr, raw ptrs, result of find (?)
- maybe_view
  - Part range (can iterate over, etc.)
  - Part optional (at most 1 value)
- Give us the basis to implement anything!

A Short Case Study



TechAtBloomberg.com

Bloomberg

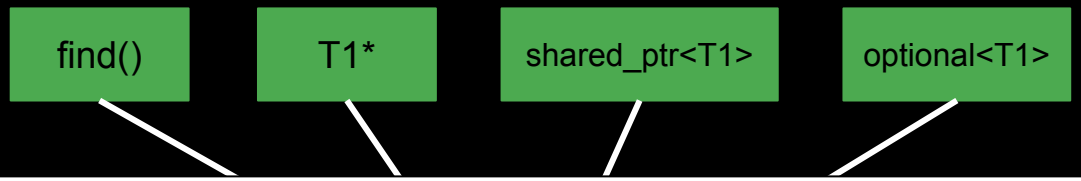
Engineering

Revisiting nullable(s) again

- Maybe-like types can be viewed as a range
  - **The whole expressiveness of ranges algorithms**
- More intuitive to leave them as they are?
  - e.g., begin(), end()?
  - e.g., size() == 0 instead of has_value()?
  - Need to convert to the range protocol & then back to the maybe-like protocol
- Just as we have range concept + algorithms
  - Can we not have the same for maybe-like types?

Revisiting nullable(s) again

- Maybe-like types can be viewed as a range
  - **The whole expressiveness of ranges algorithms**
- More intuitive to leave them as they are?
  - e.g., begin(), end()?
  - e.g., size() == 0 instead of has_value()?
  - Need to convert to the range protocol & then back to the maybe-like protocol
- Just as we have range concept + algorithms
  - Can we not have the same for maybe-like types?




```
01: template <typename Nullable>
02: class nullable_view<Nullable> : public ranges::view_interface<nullable_view<Nullable>> {
03: public:
04:     constexpr nullable_view() : value_(nullptr){};
05:
06:     constexpr explicit nullable_view(Nullable& nullable)
07:     : value_(std::addressof(nullable)) {}
08:     // ...more construction...
09:
10:     constexpr const T* data() const noexcept { /* implementation */ }
11:
12:     constexpr T*       begin() noexcept { return data(); }
13:     constexpr const T* begin() const noexcept { return data(); }
14:
15:     constexpr T*       end() noexcept { return data() + size(); }
16:     constexpr const T* end() const noexcept { return data() + size(); }
17:
18:     constexpr size_t size() const noexcept {
19:         if (!value_)
20:             return 0;
21:         return 1;
22:     }
23: private:
24:     Nullable* value_;
25: };
```

Conversion(s)

maybe::to<std::optional>

std::shared_ptr<T2>

How does this look?

```
1: std::optional<std::string> doubleNumbers(
2:     const std::optional<std::string>& input)
3: {
4:     return input
5:         | maybe::views::and_then(convertStringToInt)
6:         | maybe::views::transform(doubleInt)
7:         | maybe::views::transform(convertIntToString)
8:         | maybe::to<std::optional>;
9: }
```

```
1: std::unique_ptr<std::string> doubleNumbers(
2:     const std::unique_ptr<std::string>& input)
3: {
4:     return input
5:         | maybe::views::and_then(convertStringToInt)
6:         | maybe::views::transform(doubleInt)
7:         | maybe::views::transform(convertIntToString)
8:         | maybe::to<std::unique_ptr>;
9: }
```

```
1: std::unique_ptr<std::string> doubleNumbers(
2:     std::string* input)
3: {
4:     return input
5:         | maybe::views::and_then(convertStringToInt)
6:         | maybe::views::transform(doubleInt)
7:         | maybe::views::transform(convertIntToString)
8:         | maybe::to<std::unique_ptr>;
9: }
```

```
1: std::optional<std::string> doubleNumbers(
2:     const std::unordered_map<std::string, int>& inpMap,
3:     const std::string& searchKey)
4: {
5:     return maybe::views::find(inpMap, searchKey)  ????
6:         | maybe::views::transform(doubleInt)
7:         | maybe::views::transform(convertIntToString)
8:         | maybe::to<std::optional>
9: }
```

Behind the scenes (a sneak-peak)

```
01: // namespace maybe
02:
03: template <typename T>
04: concept not_void = !std::same_as<T, void>;
05:
06: template <typename T>
07: concept maybe_like = requires (T obj) {
08:     {bool(obj)}; // Check for presence/absence
09:     {*obj} -> not_void; // Obtain the stored element
10: };
11:
12:
13:
14:
15:
16:
```

Behind the scenes (a sneak-peak)

```
01: // namespace maybe
02:
03: template <typename T>
04: concept not_void = !std::same_as<T, void>;
05:
06: template <typename T>
07: concept maybe_like = requires (T obj) {
08:     {bool(obj)}; // Check for presence/absence
09:     {*obj} -> not_void; // Obtain the stored element
10: };
11:
12: template <typename Derived>
13: struct view_interface {
14:     template <maybe_like A = Derived>
15:     explicit constexpr view_interface(A* = nullptr) {}
16: };
```

```
01: // namespace maybe
02:
03: template <typename InputView, typename Callable>
04: requires /* InputView is maybe_like && Callable is invocable
05:             with decltype(*std::declval<InputView>) && more */
06: class transform_view
07:     : public view_interface<transform_view<Maybe, Callable>> {
08: public:
09:     constexpr transform_view(InputView inputView, Callable callable)
10:     : /* store safely */ {}
11:
12:     constexpr operator bool() { return bool(d_inputView); }
13:
14:     constexpr decltype(auto) operator*() {
15:         return d_callable(*d_inputView);
16:     }
17: private:
18:     /* storage-box-type */ d_inputView;
19:     /* storage-box-type */ d_callable;
20: };
```

```
1: // namespace maybe::views
2:
3: template<typename Callable>
4: constexpr /* maybe adaptor closure*/ transform(Callable&& c);
```

Behind the scenes (a sneak-peak)

```
01: // namespace maybe
02:
03: template <typename T>
04: concept not_void = !std:
05:
06: template <typename T>
07: concept maybe_like = req
08:     {bool(obj)}; // Chec
09:     {*obj} -> not_void;
10: };
11:
12: template <typename Deriv
13: struct view_interface {
14:     template <maybe_like A = Derived>
15:     explicit constexpr view_interface(A* = nullptr) {}
16: };
```

```
1: auto closure = maybe::views::transform(f1)
2:     | maybe::views::and_then(f2)
3:     | maybe::views::or_else(f3);
4:
5: /* some-maybe-type */ mbObj {/* initialization */};
6: auto res = mbobj | closure;  res is our or_else view
7: std::cout << "res=" << *res << std::endl;
8:
9: auto asOpt = res | maybe::to<std::optional>;
```

```
01: // namespace maybe
02:
03: template <typename InputView, typename Callable>
04: requires /* InputView is maybe_like && Callable is invocable
05:             with decltype(*std::declval<InputView>) && more */
06: class transform_view
07:     : public view_interface<transform_view<Maybe, Callable>> {
                                                    ew inputView, Callable callable)
                                                    rn bool(d_inputView); }
                                                    r*() {
                                                    ew);
                                                    ew;
19:     /* storage-box-type */ d_callable;
20: };
```

```
1: // namespace maybe::views
2:
3: template<typename Callable>
4: constexpr /* maybe adaptor closure*/ transform(Callable&& c);
```

What about almost maybe-like types?

```
01: // namespace maybe
02:
03: template <typename T>
04: class future_view
05:     : public view_interface<maybe_future_view<T>> {
06: public:
07:     explicit future_view(std::future<T>& future)
08:     : /* store safely */ {}
09:
10:     operator bool() { return d_futureStorage->valid(); }
11:
12:     decltype(auto) operator*() { return d_futureStorage->get(); }
13: private:
14:     /* storage-box-type */ d_futureStorage;
15: };
```

```
1: // namespace maybe::views
2: template <typename T>
3: constexpr /* view-type */ future(std::future<T>& future);
```

```
1: std::optional<std::string> doubleNumbers(
2:     std::future<std::string>& input)
3: {
4:     return maybe::views::future(input)
5:         | maybe::views::and_then(convertStringToInt)
6:         | maybe::views::transform(doubleInt)
7:         | maybe::views::transform(convertIntToString)
8:         | maybe::to<std::optional>;
9: }
```

Additional notes

- More algorithms/views possible
  - **find()**, **and_then()**, **transform_or()**, etc.
  - Exclusively for the maybe protocol!
- **operator|()** has 2 flavors:
  - If **C** and **D** are closure objects → **C | D** produces another closure object
  - If **M** is maybe object and **C** is a closure object → **M | C** produces a lazy maybe view
- **maybe::to** conversion
  - What can it take?
  - **std::optional** for sure, but what about **std:unique_ptr**/**std::shared_ptr**, raw ptrs?
- Refine the storage requirements for our views
  - What/when/how do we “retain”?
  - e.g., **transform** retains the callable by value, but the input maybe as a “view”
  - e.g., the “caching” of lazy results can be retained via **std::optional**
  - As much as possible mimic what ranges does…
- Is **null** the only alternative?
  - e.g., if **std::expected** is to be supported, we need to customize the alternative case (e.g., exception instead of null)
- Do we just produce results?

Non-result Producing Algorithms

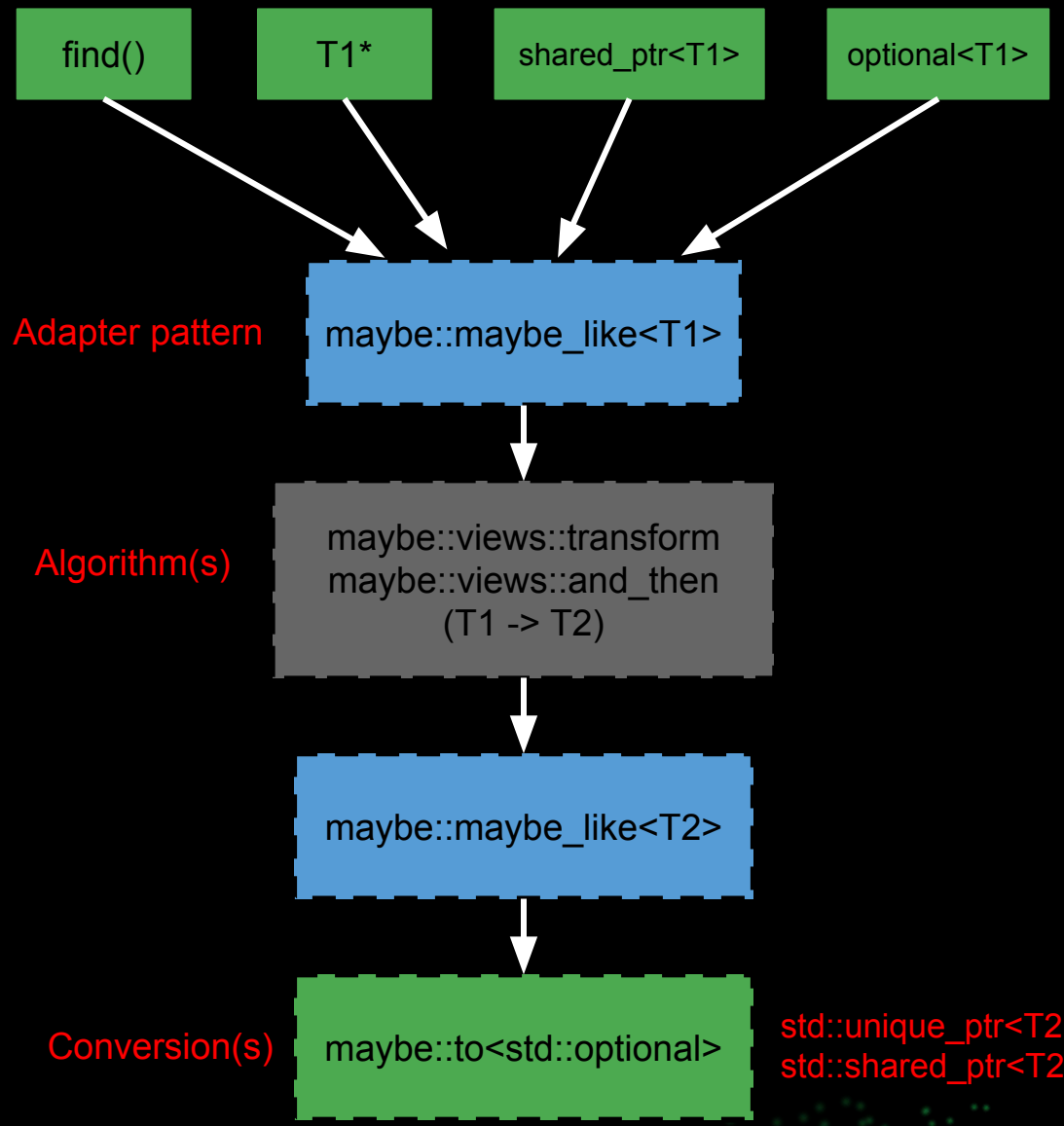
```
1: std::optional o{7}; // with p1255
2: for (auto&& i : views::nullable(std::ref(o))) {
3:     i = 9;
4:     std::cout << "i=" << i << " prints 9\n";
5: }
6: std::cout << "o=" << *o << " prints 9\n";
```

VS.

```
1: std::optional o{7}; // Also *s, smart ptrs, etc.
2: maybe::unwrap(o, [](auto&& i){
3:     i = 9;
4:     std::cout << "i=" << i << " prints 9\n";
5: });
6: std::cout << "o=" << *o << " prints 9\n";
```

Revisiting nullable(s) again

- Maybe-like types can be viewed as a range
  - **The whole expressiveness of ranges algorithms**
- More intuitive to leave them as they are?
  - e.g., begin(), end()?
  - e.g., size() == 0 instead of has_value()?
  - Need to convert to the range protocol & then back to the maybe-like protocol
- Just as we have range concept + algorithms
  - Can we not have the same for maybe-like types?



Thank you!

https://TechAtBloomberg.com/cplusplus

https://www.bloomberg.com/careers

Contact us: gkoyrushki@bloomberg.net
afisher92@bloomberg.net

Bloomberg

Engineering

TechAtBloomberg.com

Appendix A.1: Optional Functor Laws

Adopted to C++ from http://learnyouahaskell.com/functors-applicative-functors-and-monoids

```
01: int val = 1;
02: std::optional<int> myOpt {val};
03:
04: // LAW 1: Identity Preservation
05: auto id = [](int el) { return el; };
06: assert(
07:     myOpt.transform(id) IS_EQUIV myOpt
08: );
09:
10: // LAW 2: Composition Preservation
11: auto f = [](int el) { return el * 2; };
12: auto g = [](int el) { return el - 2; };
13: assert(
14:     myOpt.transform(f).transform(g) IS_EQIV
15:     myOpt.transform([](int el) { return g(f(el)); })
16: );
```

Appendix A.2: Optional Monad Laws

Adopted to C++ from http://learnyouahaskell.com/a-fistful-of-monads

```
01: // Haskell calls it return
02: auto wrap = [](int el) { return std::optional<int>{el}; };
03:
04: int val = 1;
05: auto mul2 = [](int el) { return std::optional<int>{el * 2}; };
06:
07: // LAW 1: Left Identity Preservation
08: assert(
09:     wrap(val).and_then(mul2) IS_EQUIV mul2(val)
10: );
11:
12: // LAW 2: Right Identity Preservation
13: std::optional<int> myOpt {val};
14: assert(
15:     myOpt.and_then(wrap) IS_EQUIV myOpt;
16: );
17:
18: // LAW 3: Associativity Preservation
19: auto f = [](int el) { return std::optional<int>{el * 2}; };
20: auto g = [](int el) { return std::optional<int>{el - 2}; };
21: assert(
22:     // Haskell notation
23:     // (myOpt >= f) >= g IS_EQUIV myOpt >>= (\x -> f x >>= g)
24:     myOpt.and_then(f).and_then(g) IS_EQIV
25:     myOpt.and_then([](int el){ return f(el).and_then(g); })
26: );
```

And the Algorithms?

```
01: // namespace maybe::detail
02:
03: template <typename SpecificAlgorithm>
04: struct Algorithm {
05:     template <typename Maybe>
06:     requires maybe_like<Maybe>
07:     friend constexpr auto operator|(Maybe&& maybe, Algorithm&& algorithm) {
08:         return std::invoke(static_cast<SpecificAlgorithm&&>(algorithm), std::forward<Maybe>(maybe));
09:     }
10: };
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
```

!!! This is eager evaluation

CRTP: each algorithm defines its own () method

And the Algorithms?

!!! This is eager evaluation
(not too hard to make it lazy)

CRTP: each algorithm
defines its own () method

```
01: // namespace maybe::detail
02:
03: template <typename SpecificAlgorithm>
04: struct Algorithm {
05:     template <typename Maybe>
06:     requires maybe_like<Maybe>
07:     friend constexpr auto operator|(Maybe&& maybe, Algorithm&& algorithm) {
08:         return std::invoke(static_cast<SpecificAlgorithm&&>(algorithm), std::forward<Maybe>(maybe));
09:     }
10: };
11:
12: template <typename Callable>
13: struct TransformAlgorithm : public Algorithm<TransformAlgorithm<Callable>> {
14:     Callable d_callable;
15:
16:     template <typename Maybe>
17:     constexpr auto operator()(Maybe&& maybe) const { /* invoke d_callable on maybe */ }
18: };
19:
20: template <typename Callable>
21: struct AndThenAlgorithm : public Algorithm<AndThenAlgorithm<Callable>> {
22:     Callable d_callable;
23:
24:     template <typename Maybe>
25:     constexpr auto operator()(Maybe&& maybe) const { /* invoke d_callable on maybe */ }
26: };
```

And the Algorithms?

```
01: // namespace maybe::detail
02:
03: template <template <typename> class SpecificAlgorithm>
04: struct AlgorithmCreator {
05:     template <typename Callable>
06:     constexpr auto operator()(Callable&& c) const {
07:         return SpecificAlgorithm<Callable&&>{{}, std::forward<Callable>(c)};
08:     }
09: };
```

inputOpt | maybe::transform([](auto && el) {})

And the Algorithms?

```
01: // namespace maybe::detail
02:
03: template <template <typename> class SpecificAlgorithm>
04: struct AlgorithmCreator {
05:     template <typename Callable>
06:     constexpr auto operator()(Callable&& c) const {
07:         return SpecificAlgorithm<Callable&&>{{}, std::forward<Callable>(c)};
08:     }
09: };
```

inputOpt | maybe::transform([](auto && el) {})

```
01: // namespace maybe::views
02: constexpr static auto and_then     = detail::AlgorithmCreator<detail::AndThenAlgorithm>();
03: constexpr static auto transform    = detail::AlgorithmCreator<detail::TransformAlgorithm>();
04: // other algorithms: or_else, transform_or, value_or, etc., etc., etc.
```

And the Algorithms?

```
01: // namespace maybe::detail
02:
03: template <template <typename> class SpecificAlgorithm>
04: struct AlgorithmCreator {
05:     template <typename Callable>
06:     constexpr auto operator()(Callable&& c) const {
07:         return SpecificAlgorithm<Callable&&>{{}, std::forward<Callable>(c)};
08:     }
09: };
```

```
01: std::optional<std::string> doubleNumbers(
02:     const std::optional<std::string>& input)
03: {
04:     return input
05:         | maybe::views::and_then(convertStringToInt)
06:         | maybe::views::transform(doubleInt)
07:         | maybe::views::transform(convertIntToString)
08:         | maybe::to<std::optional>;
09: }
```

```
01: // namespace maybe::views
02: constexpr static auto and_then     = detail::AlgorithmCreator<detail::AndThenAlgorithm>();
03: constexpr static auto transform    = detail::AlgorithmCreator<detail::TransformAlgorithm>();
04: // other algorithms: or_else, transform_or, value_or, etc., etc., etc.
```