Back To Basics
Functional Programming in C++

JONATHAN MÜLLER

Cppcon
The C++ Conference

2024
September 15 - 20

What is functional programming?

Imperative Programming

Definition

Specify instructions that manipulate state in order to achieve a goal.

Imperative Programming

Definition

Specify instructions that manipulate state in order to achieve a goal.

- C and C++

think-cell

Imperative Programming

Definition

Specify instructions that manipulate state in order to achieve a goal.

- C and C++
- CPU

Imperative Programming

Definition

Specify instructions that manipulate state in order to achieve a goal.

- C and C++
- CPU
- IKEA manual

think-cell

Declarative Programming

Definition

Specify the desired outcome (only), have the system figure it out how to achieve it.

Declarative Programming

Definition

Specify the desired outcome (only), have the system figure it out how to achieve it.

- Haskell, Prolog

Declarative Programming

Definition

Specify the desired outcome (only), have the system figure it out how to achieve it.

- Haskell, Prolog
- formal grammar

think-cell

Declarative Programming

Definition

Specify the desired outcome (only), have the system figure it out how to achieve it.

- Haskell, Prolog
- formal grammar
- thermostat

think-cell

Functional Programming

Definition

Declarative programming by composing functions.

Functional Programming

Definition

Declarative programming by composing functions.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

Functional Programming

Definition

Declarative programming by composing functions.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = sort left ++ [x] ++ sort right
    where
        left  = filter (<  x) xs
        right = filter (>= x) xs
```

think-cell

Why functional programming?

No state: easier to reason about, easier to parallelize

Functional Programming

Why functional programming?

No state: easier to reason about, easier to parallelize

Why not functional programming?

Less efficient

But C++ isn't a pure functional language!

But C++ isn't a pure functional language!

We don't need to use functional programming for everything.

Functional programming in C++

Definition

Declarative programming by composing functions.

Definition

Declarative programming by composing functions.

Functional programming is all about COMPOSITION.

think-cell

Functional programming in C++

1. Write building blocks using efficient, optimized, and tested (!) C++ code.
2. Compose building blocks using functional paradigms.

Composing algorithms

Example: Biggest odd magnitude

Problem statement

Input Non-empty list of integers

Output The biggest magnitude of an odd integer in the list

think-cell

Example: Biggest odd magnitude

Problem statement

Input Non-empty list of integers
Output The biggest magnitude of an odd integer in the list

Input `[3, 0, 2, -1, 5, -7, 8]`
Output `7`

Example: Biggest odd magnitude

```cpp
int biggest_odd_magnitude(auto&& rng) {
    int candidate = -1;
    for (int x : rng) {
        int magnitude = std::abs(x);
        if (magnitude % 2 == 1) {
            if (magnitude > candidate) {
                candidate = magnitude;
            }
        }
    }
    return candidate;
}
```

Example: Biggest odd magnitude

What is the composition?

Example: Biggest odd magnitude

What is the composition?

1. Compute the magnitude of each number.
2. Filter out even numbers, keep only odd numbers.
3. Find the maximum.

think-cell

Example: Biggest odd magnitude

```cpp
int biggest_odd_magnitude(auto&& rng) {
    // 1. Compute the magnitude of each number.
    std::vector<int> magnitudes;
    stdr::transform(rng, std::back_inserter(magnitudes),
                           [](int x) { return std::abs(x); });

    // 2. Filter out even numbers, keep only odd numbers.
    std::vector<int> odd_magnitudes;
    stdr::copy_if(magnitudes, std::back_inserter(odd_magnitudes),
                         [](int x) { return x % 2 == 1; });

    // 3. Find the maximum.
    return stdr::max(odd_magnitudes);
}
```

New `std::ranges` algorithm

Old:

```
std::transform(std::begin(rng), std::end(rng), std::back_inserter(squares),
               [](int x) { return x * x; });
```

New:

```
namespace stdr = std::ranges;

stdr::transform(rng, std::back_inserter(squares),
                      [](int x) { return x * x; });
```

think-cell

Problem: Eager composition

- We have to create intermediate containers.

Problem: Eager composition

- We have to create intermediate containers.
- We eagerly compute all intermediate results, even if unneeded.

think-cell

Problem: Eager composition

- We have to create intermediate containers.
- We eagerly compute all intermediate results, even if unneeded.
- We manually deal with intermediate state.

think-cell

Solution: Lazy composition

View: A lazy range whose values are computed on demand.

```
namespace stdv = std::views;
```

Solution: Lazy composition

View: A lazy range whose values are computed on demand.

```
namespace stdv = std::views;
```

- Composing views does nothing yet.

Solution: Lazy composition

View: A lazy range whose values are computed on demand.

```
namespace stdv = std::views;
```

- Composing views does nothing yet.
- Only iteration will trigger computation.

Solution: Lazy composition

View: A lazy range whose values are computed on demand.

```
namespace stdv = std::views;
```

- Composing views does nothing yet.
- Only iteration will trigger computation.
- Functional pipeline style; no manual state wrangling.

Example: Biggest odd magnitude

```cpp
int biggest_odd_magnitude(auto&& rng) {
    // 1. Compute the magnitude of each number.
    auto magnitudes = rng
        | stdv::transform([](int x) { return std::abs(x); });
    // 2. Filter out even numbers, keep only odd numbers.
    auto odd_magnitudes = magnitudes
        | stdv::filter([](int x) { return x % 2 == 1; });
    // 3. Find the maximum.
    return stdr::max(odd_magnitudes);
}
```

Example: Biggest odd magnitude

```
int biggest_odd_magnitude(auto&& rng) {
    return stdr::max(
        rng
        | stdv::transform([](int x) { return std::abs(x); })
        | stdv::filter([](int x) { return x % 2 == 1; })
    );
}
```

Composition shape: Range $\rightarrow$ Range

Input Range
Output Range with different values or a different size

Composition shape: Range $\to$ Range

Input Range
Output Range with different values or a different size

- `stdv::transform`: apply a function to each element ("map")

think-cell

Composition shape: Range $\to$ Range

Input Range
Output Range with different values or a different size

- `stdv::transform`: apply a function to each element (“map”)
- `stdv::filter`: keep only elements that satisfy a predicate

think-cell

Composition shape: Range → Range

Input Range

Output Range with different values or a different size

- `stdv::transform`: apply a function to each element (“map”)
- `stdv::filter`: keep only elements that satisfy a predicate
- `stdv::take_while`/`drop_while`: keep only the first/last elements

think-cell

Composition shape: Range $\rightarrow$ derived value

Input	Range
Output	Derived value

Composition shape: Range $\longrightarrow$ derived value

Input Range
Output Derived value

- `stdr::max*/min*`: find the maximum/minimum element

think-cell

Composition shape: Range $\rightarrow$ derived value

Input Range
Output Derived value

- `stdr::max*`/`min*`: find the maximum/minimum element
- `stdr::count*`: count the number of elements satisfying a predicate

think-cell

Composition shape: Range $\rightarrow$ derived value

Input Range
Output Derived value

- `stdr::max*/min*`: find the maximum/minimum element
- `stdr::count*`: count the number of elements satisfying a predicate
- `stdr::all_of/any_of/none_of`: check if all/any/none of the elements satisfy a predicate

think-cell

Composition shape: Range $\longrightarrow$ derived value

Input Range
Output Derived value

- `stdr::max*/min*`: find the maximum/minimum element
- `stdr::count*`: count the number of elements satisfying a predicate
- `stdr::all_of/any_of/none_of`: check if all/any/none of the elements satisfy a predicate
- `stdr::find*`: iterator to element satisfying some condition

Composition shape: Range → derived value

Input Range
Output Derived value

- `stdr::max*/min*`: find the maximum/minimum element
- `stdr::count*`: count the number of elements satisfying a predicate
- `stdr::all_of/any_of/none_of`: check if all/any/none of the elements satisfy a predicate
- `stdr::find*`: iterator to element satisfying some condition
- `stdr::search*`: view of subrange matching some other range

Example: Smallest Fibonacci number above a threshold

Problem statement

Input	Threshold x
Output	The smallest Fibonacci number greater than x

Example: Smallest Fibonacci number above a threshold

Problem statement

Input Threshold x

Output The smallest Fibonacci number greater than x

Input 9

Output 13 (`fib(6) = 8`, `fib(7) = 13`)

think-cell

Example: Smallest Fibonacci number above a threshold

What is the composition?

Example: Smallest Fibonacci number above a threshold

What is the composition?

1. Generate all Fibonacci numbers.
2. Find the first one greater than x.

Example: Smallest Fibonacci number above a threshold

```cpp
int smallest_fib_above(int x) {
    auto all_fibonacci_numbers = …;



    return *stdr::find_if(
        all_fibonacci_numbers,
        [x](int f) { return f > x; }
    );
}
```

think-cell

Example: Smallest Fibonacci number above a threshold

```cpp
int fib(int n) { … }

int smallest_fib_above(int x) {
    auto all_fibonacci_numbers =
        stdv::iota(0)              // [0, 1, 2, 3, ...]
        | stdv::transform(fib); // [0, 1, 1, 2, ...]

    return *stdr::find_if(
        all_fibonacci_numbers,
        [x](int f) { return f > x; }
    );
}
```

think-cell

Example: Smallest Fibonacci number above a threshold

```cpp
int fib(int n) {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
```

Example: Smallest Fibonacci number above a threshold

```cpp
int fib(int n) {
    int a = 0, b = 1;
    for (auto _ : stdv::iota(0, n)) {
        auto c = a + b;
        a = b;
        b = c;
    }
    return a;
}
```

think-cell

Example: Smallest Fibonacci number above a threshold

```cpp
std::generator<int> all_fibonacci_numbers() {
    int a = 0, b = 1;
    while (true) {
        co_yield a;

        auto c = a + b;
        a = b;
        b = c;
    }
}
```

```cpp
int smallest_fib_above(int x) {
    return *stdr::find_if(
        all_fibonacci_numbers(),
        [x](int f) { return f > x; }
    );
}
```

Composition shape: Nothing → Range

Input	Nothing
Output	Range

Composition shape: Nothing → Range

Input	Nothing
Output	Range

- `stdv::iota`: generate an incrementing range of numbers

Composition shape: Nothing → Range

Input Nothing
Output Range

- `stdv::iota`: generate an incrementing range of numbers
- `stdv::repeat`: generate an infinite range of the same value

Composition shape: Nothing → Range

Input Nothing
Output Range

- `stdv::iota`: generate an incrementing range of numbers
- `stdv::repeat`: generate an infinite range of the same value
- `std::generator` + `co_yield`: generate an arbitrary range of values

Example: Find sum of two integers

Problem statement

Input	Two lists of integers, a number x
Output	The two integers from the list whose sum is x

Example: Find sum of two integers

Problem statement

Input Two lists of integers, a number x

Output The two integers from the list whose sum is x

Input `[1, 2, 3, 4]`, `[5, 6, 7, 8]`, `10`

Output `(2, 8)`

think-cell

Example: Find sum of two integers

What is the composition?

Example: Find sum of two integers

What is the composition?

1. Generate all pairs.
2. Find the pair with the sum x.

Example: Find sum of two integers

```
auto find_sum(auto&& rng1, auto&& rng2)
  -> std::optional<std::tuple<int, int>>
{
    auto combinations = stdv::cartesian_product(rng1, rng2);
    auto iter = stdr::find_if(combinations, [x](auto p) {
        return std::get<0>(p) + std::get<1>(p) == x;
    });
    return iter == combinations.end()
        ? std::nullopt
        : std::make_tuple(std::get<0>(*iter), std::get<1>(*iter));
}
```

Aside: think-cell-library

```
auto find_sum(auto&& rng1, auto&& rng2)
  -> std::optional<std::tuple<int, int>>
{
    auto combinations = stdv::cartesian_product(rng1, rng2);
    return tc::find_first_if<tc::return_value_or_none>(
        combinations,
        [x](auto a, auto b) {
            return a + b == x;
        }
   );
}
```

think-cell

Composition shape: Multiple ranges → single range

Input Multiple ranges
Output Single range

think-cell

Composition shape: Multiple ranges → single range

Input Multiple ranges

Output Single range

- `stdv::cartesian_product`: `std::tuple` of all possible element combinations (`N * M` elements)

Composition shape: Multiple ranges → single range

Input Multiple ranges
Output Single range

- `stdv::cartesian_product`: `std::tuple` of all possible element combinations (`N * M` elements)
- `stdv::zip`: `std::tuple` of corresponding elements (`min(N, M)` elements)

Composition shape: Multiple ranges → single range

Input Multiple ranges
Output Single range

- `stdv::cartesian_product`: `std::tuple` of all possible element combinations (`N * M` elements)
- `stdv::zip`: `std::tuple` of corresponding elements (`min(N, M)` elements)
- `stdv::concat`: all elements of each range in order (`N + M` elements)

think-cell

Example: Longest contiguous subsequence of increasing numbers

Problem statement

Input	List of integers
Output	The length of longest contiguous subsequence of increasing numbers

Example: Longest contiguous subsequence of increasing numbers

Problem statement

Input List of integers

Output The length of longest contiguous subsequence of increasing numbers

Input `[1, 3, 2, 4, 5, 7, 6]`

Output `4` (`[2, 4, 5, 7]`)

Example: Longest contiguous subsequence of increasing numbers

What is the composition?

Example: Longest contiguous subsequence of increasing numbers

What is the composition?

1. Split the list into multiple chunks of increasing numbers.
2. Determine the size of each chunk.
3. Return the maximum.

think-cell

Example: Longest contiguous subsequence of increasing numbers

```
std::size_t longest_increasing_subsequence(auto&& rng)
{
    return stdr::max(
        rng                                 // [1, 3, 2, 4, 5, 7, 6]
        | stdv::chunk_by(std::less<int>{}) // [[1, 3], [2, 4, 5, 7], [6]]
        | stdv::transform(stdr::size)      // [2, 4, 1]
    );
}
```

think-cell

Composition shape: Range $\longrightarrow$ range of ranges

Input	Range
Output	Range of ranges

Composition shape: Range → range of ranges

Input Range
Output Range of ranges

- `stdv::chunk_by`: split the range into chunks where borders don't satisfy a binary predicate

Composition shape: Range $\longrightarrow$ range of ranges

Input Range
Output Range of ranges

- `stdv::chunk_by`: split the range into chunks where borders don't satisfy a binary predicate
- `stdv::split`: split a range at a separator

think-cell

Composition shape: Range → range of ranges

Input Range
Output Range of ranges

- `stdv::chunk_by`: split the range into chunks where borders don't satisfy a binary predicate
- `stdv::split`: split a range at a separator
- `stdv::adjacent/slide`: a range of sliding windows into the input range

Example: Reverse words

Problem statement

Input A string

Output The string where all words (sequences of letters) are reversed

Example: Reverse words

Problem statement

Input A string

Output The string where all words (sequences of letters) are reversed

Input “Hello World!”

Output “olleH dlroW!”

Example: Reverse words

What is the composition?

Example: Reverse words

What is the composition?

1. Split the string into words.
2. Reverse each chunk that is a word.
3. Put all the chunks back together.

Example: Reverse words

```cpp
std::string reverse_words(std::string const& str)
{
    return str // "Hello World!"
      | stdv::chunk_by([](char left, char right) {
          return std::isalpha(left) != std::isalpha(right);
      }) // ["Hello", " ", "World", "!"]
      | stdv::transform([](auto chunk) {
          if (std::isalpha(chunk.front()))
              return chunk | stdv::reverse | stdr::to<std::string>();
          else
              return chunk | stdr::to<std::string>();
      }) // ["olleH", " ", "dlroW", "!"]
      | stdv::join | stdr::to<std::string>(); // "olleH dlroW!"
}
```

Example: Reverse words

```cpp
std::string reverse_words(std::string const& str)
{
    return str // "Hello World!"
      | stdv::chunk_by([](char left, char right) {
          return std::isalpha(left) != std::isalpha(right);
      }) // ["Hello", " ", "World", "!"]
      | stdv::transform([](auto chunk) {
          if (std::isalpha(chunk.front()))
              return chunk | stdv::reverse | stdr::to<std::string>();
          else
              return chunk | stdr::to<std::string>();
      }) // ["olleH", " ", "dlroW", "!"]
      | stdv::join | stdr::to<std::string>(); // "olleH dlroW!"
}
```

Example: Reverse words

```cpp
std::string reverse_words(std::string const& str)
{
    return str // "Hello World!"
      | stdv::chunk_by([](char left, char right) {
          return std::isalpha(left) != std::isalpha(right);
      }) // ["Hello", " ", "World", "!"]
      | stdv::transform([](auto chunk) {
          return tc_conditional_range(
              std::isalpha(chunk.front()),
                  chunk | stdv::reverse,
                  chunk
          );
      }) // ["olleH", " ", "dlroW", "!"]
      | stdv::join | stdr::to<std::string>(); // "olleH dlroW!"
}
```

Composition shape: Range of ranges → Single range

Input	Range of ranges
Output	Single range

Composition shape: Range of ranges → Single range

Input Range of ranges
Output Single range

- `stdv::join`: flattens a range of ranges

Composition shape: Range of ranges → Single range

Input Range of ranges
Output Single range

- `stdv::join`: flattens a range of ranges
- `stdv::join_with`: flattens a range of ranges, inserting a separator between them

General takeways

- Try to identify small building blocks for solving a problem

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`
  - Joining ranges: `join`, `join_with`

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`
  - Joining ranges: `join`, `join_with`
- Know your algorithms:

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`
  - Joining ranges: `join`, `join_with`
- Know your algorithms:
  - Folds: `max*/min*`, `count*`, `all_of/any_of/none_of`

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`
  - Joining ranges: `join`, `join_with`
- Know your algorithms:
  - Folds: `max*/min*`, `count*`, `all_of/any_of/none_of`
  - Searches: `find*`, `search*`

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`
  - Joining ranges: `join`, `join_with`
- Know your algorithms:
  - Folds: `max*/min*`, `count*`, `all_of/any_of/none_of`
  - Searches: `find*`, `search*`

think-cell

General takeways

- Try to identify small building blocks for solving a problem
  - Creating ranges: `iota`, `generator`
  - Transforming ranges: `transform`, `filter`
  - Combining ranges: `cartesian_product`, `zip`, `concat`
  - Splitting ranges: `split`, `chunk_by`
  - Joining ranges: `join`, `join_with`
- Know your algorithms:
  - Folds: `max*/min*`, `count*`, `all_of/any_of/none_of`
  - Searches: `find*`, `search*`

Composing algorithms minimizes use of `for`.

think-cell

Composing functions

Composing functions

```
void unsubscribe_from_mailing_list(userID id) {
  subscriber_list.remove(
      lookup_user(id).email
  );
}
```

Composing functions

```
void unsubscribe_from_mailing_list(userID id) {
  subscriber_list.remove(
      lookup_user(id).email
  );
}
```

Composition chain:

```
UserID -> User -> Email
```

What if the intermediate functions can fail?

std::optional

```
std::optional<User> lookup_user(UserID id);
```

think-cell

std::optional

```
std::optional<User> lookup_user(UserID id);
```

```
std::optional<T>
```

Either holds a T or is empty.

Composing optional functions

```
void unsubscribe_from_mailing_list(UserID id) {
    std::optional<User> user = lookup_user(id);
    if (user) {
        subscriber_list.remove(user->email);
    }
}
```

Composing optional functions

```cpp
void unsubscribe_from_mailing_list(UserID id) {
    std::optional<User> user = lookup_user(id);
    if (user) {
        subscriber_list.remove(user->email);
    }
}
```

Composition chain:

`UserID -> User or nothing -> Email or nothing`

Composing optional functions

Given a `std::optional<User>` how do I get a `std::optional<Email>`?

Composing optional functions

Given a `std::optional<User>` how do I get a `std::optional<Email>`?

```
auto email = user ? std::make_optional(user->email) : std::nullopt;
```

think-cell

std::optional::transform

```
// f has signature: (T) -> U
auto std::optional<T>::transform(auto&& f) {
    return *this
        ? std::make_optional(f(this->value()));
        : std::nullopt;
}
```

std::optional::transform

```cpp
// f has signature: (T) -> U
auto std::optional<T>::transform(auto&& f) {
    return *this
        ? std::make_optional(f(this->value()));
        : std::nullopt;
}
```

```cpp
auto email = user.transform(&User::email);
```

Composing optional functions

Given a `std::optional<Email>` how do I invoke `subscriber_list.remove`?

Composing optional functions

Given a `std::optional<Email>` how do I invoke `subscriber_list.remove`?

```
if (email) {
    subscriber_list.remove(*email);
}
```

Composing optional functions

Given a `std::optional<Email>` how do I invoke `subscriber_list.remove`?

```
email.transform([&](Email const& e) {
    subscriber_list.remove(e);
});
```

think-cell

Composing optional functions

Given a `std::optional<Email>` how do I invoke `subscriber_list.remove`?

```
email.transform([&](Email const& e) {
    subscriber_list.remove(e);
    return std::monostate{};
});
```

Composing optional functions

```
void unsubscribe_from_mailing_list(UserID id) {
    lookup_user(id)                             // User
        .transform(&User::email)                // Email
        .transform([&](Email const& e) { // action
            subscriber_list.remove(e);
            return std::monostate{};
        });
}
```

Composing optional functions with other optional functions

What if not every user has an email address?

```cpp
struct User {
    std::optional<Email> email;
};
```

Composing optional functions with other optional functions

What if not every user has an email address?

```
struct User {
    std::optional<Email> email;
};
```

```
std::optional<std::optional<Email>> email
    = user.transform(&User::email);
if (email) { // We have a user.
    if (*email) { // The user has an email.
        subscriber_list.remove(**email);
    }
}
```

think-cell

Flatten a nested optional

```
template <typename T>
std::optional<T> join(std::optional<std::optional<T>> const& opt_opt_t) {
    if (opt_opt_t)
        return *opt_opt_t;
    else
        return std::nullopt;
}
```

Flatten a nested optional

```cpp
template <typename T>
std::optional<T> join(std::optional<std::optional<T>> const& opt_opt_t) {
    if (opt_opt_t)
        return *opt_opt_t;
    else
        return std::nullopt;
}
```

```cpp
auto email = join(user.transform(&User::email));
```

std::optional::and_then

```cpp
// f has signature: (T) -> optional<U>
auto std::optional<T>::and_then(auto&& f) {
    return *this
        ? f(this->value());
        : std::nullopt;
}
```

std::optional::and_then

```cpp
// f has signature: (T) -> optional<U>
auto std::optional<T>::and_then(auto&& f) {
    return *this
        ? f(this->value());
        : std::nullopt;
}
```

```cpp
auto email = user.and_then(&User::email);
```

Composing optional functions with other optional functions

```cpp
void unsubscribe_from_mailing_list(UserID id) {
    lookup_user(id)                              // User
        .and_then(&User::email)                  // Email
        .transform([&](Email const& e) { // action
            subscriber_list.remove(e);
            return std::monostate{};
        });
}
```

Composing optional functions with ranges

What if a user can have multiple email addresses?

```
struct User {
    std::vector<Email> email;
};
```

Composing optional functions with ranges

What if a user can have multiple email addresses?

```
struct User {
    std::vector<Email> email;
};
```

```
std::optional<std::vector<Email>> emails
    = user.transform(&User::email);
if (emails) {
    for (auto email : emails)
        subscriber_list.remove(email);
    }
}
```

Composing optional functions with ranges

`std::optional<T>` is a range of `0` or 1 Ts in C++26.

```
void unsubscribe_from_mailing_list(UserID id) {
    auto opt_emails = lookup_user(id)    // User
        .transform(&User::email);        // optional Emails

    stdr::for_each(
        stdv::join(opt_emails),          // Emails
        [&](Email const& e) {            // action
            subscriber_list.remove(e);
        }
    );
}
```

think-cell

Composing optional functions with ranges

`std::optional<T>` is a range of `0` or 1 Ts in C++26.

```
void unsubscribe_from_mailing_list(UserID id) {
    stdr::for_each(
        lookup_user(id)                                 // User
        | stdv::transform(&User::email) | stdv::join, // Emails
        [&](Email const& e) {                           // action
            subscriber_list.remove(e);
        }
    );
}
```

Composing optional functions with ranges

C++26/9?

`std::optional<T>` is a range of `0` or 1 Ts in C++26.

```
void unsubscribe_from_mailing_list(UserID id) {
    stdr::for_each(
        lookup_user(id)                          // User
        | stdv::transform_join(&User::email),    // Emails
        [&](Email const& e) {                    // action
            subscriber_list.remove(e);
        }
    );
}
```

Composing failible functions

```cpp
void unsubscribe_from_mailing_list(UserID id) {
    try {
        User user = lookup_user(id); // may throw DBError
        if (user) {
            subscriber_list.remove(user->email);
        }
    } catch (DBError const& e) {
        log_error(e);
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
    }
}
```

Composing failible functions

```
void unsubscribe_from_mailing_list(UserID id) {
    try {
        User user = lookup_user(id); // may throw DBError
        if (user) {
            subscriber_list.remove(user->email);
        }
    } catch (DBError const& e) {
        log_error(e);
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
    }
}
```

Composition requires *values*.

std::expected

```cpp
std::expected<User, DBError> lookup_user(UserID id);
```

std::expected

```
std::expected<User, DBError> lookup_user(UserID id);
```

```
std::expected<T, E>
```

Either holds a `T` (value) or an `E` (error).

Composing failible functions

```cpp
void unsubscribe_from_mailing_list(UserId id) {
    std::expected<User, DBError> user = lookup_user(id);
    if (user) {
        subscriber_list.remove(user->email);
    } else {
        log_error(user.error());
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
    }
}
```

Composing failible functions

```cpp
void unsubscribe_from_mailing_list(UserId id) {
    std::expected<User, DBError> user = lookup_user(id);
    if (user) {
        subscriber_list.remove(user->email);
    } else {
        log_error(user.error());
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
    }
}
```

Composition chain:

```
UserID -> User or DBError -> Email or DBError
```

std::expected::transform

```
// f has signature: (T) -> U
auto std::expected<T, E>::transform(auto&& f) {
    return *this
        ? std::expected<…, E>(f(this->value()));
        : std::expected<…, E>(std::unexpect, this->error());
}
```

think-cell

Composing failible functions

```
void unsubscribe_from_mailing_list(UserId id) {
    std::expected<void, DBError> result
      = lookup_user(id)                         // User or DBError
        .transform(&User::email)                // Email or DBError
        .transform([&](Email const& e) { // action
            subscriber_list.remove(e);
        });
    if (!result) {
        log_error(result.error());
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
    }
}
```

think-cell

std::expected::transform_error

```
// f has signature: (E) -> F
auto std::expected<T, E>::transform_error(auto&& f) {
    return *this
        ? std::expected<T, …>(this->value());
        : std::expected<T, …>(std::unexpect, f(this->error()));
}
```

Composing failible functions

```
void unsubscribe_from_mailing_list(UserId id) {
    lookup_user(id)                              // User or DBError
        .transform(&User::email)                 // Email or DBError
        .transform([&](Email const& e) { // action
            subscriber_list.remove(e);
        })
        .transform_error([&](DBError const& e) {
            log_error(e);
            retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
            return std::monostate{}; // explicit swallow
        });
}
```

Composing failible functions with other failible functions

What if the email address needs to be queried as well?

```
std::expected<Email, DBError> query_email(User const& user);
```

```
std::expected<std::expected<Email, DBError>, DBError> email
    = user.transform(&query_email);
if (email) { // We could query the user.
    if (*email) { // We could query the email.
        subscriber_list.remove(**email);
    }
}
```

think-cell

Flatten a nested expected

```cpp
template <typename T, typename E>
std::expected<T, E> join(
    std::expected<std::expected<T, E>, E> const& exp_exp_t
) {
    if (exp_exp_t)
        return *exp_exp_t;
    else
        return std::unexpected(exp_exp_t.error());
}
```

think-cell

std::expected::and_then

```cpp
// f has signature: (T) -> expected<U, E>
auto std::expected<T, E>::and_then(auto&& f) {
    return *this
        ? f(this->value());
        : std::unexpected(this->error());
}
```

Composing failible functions with other failible functions

```
void unsubscribe_from_mailing_list(UserId id) {
    lookup_user(id)                              // User or DBError
        .and_then(query_email)                   // Email or DBError
        .transform([&](Email const& e) { // action
            subscriber_list.remove(e);
        })
        .transform_error([&](DBError const& e) {
            log_error(e);
            retry_operation_later([id] { unsubscribe_from_mailing_list(id); });
            return std::monostate{}; // explicit swallow
        });
}
```

Careful: Composing `std::expected` requires the same E

```cpp
std::expected<std::string, DBError> query_value(Key key);
std::expected<int, ParseError> parse_value(std::string const& value);
```

```cpp
auto process_value(Key key)
    -> std::expected<void, ???>
{
    return query_value(key)
            .and_then(&parse_value)
            .transform([](int value) {
                std::cout << value << '\n';
            });
}
```

think-cell

Careful: Composing `std::expected` requires the same E

```cpp
std::expected<std::string, DBError> query_value(Key key);
std::expected<int, ParseError> parse_value(std::string const& value);
```

```cpp
auto process_value(Key key)
    -> std::expected<void, std::variant<DBError, ParseError>>
{
    return query_value(key)
            .and_then(&parse_value) // doesn't compile
            .transform([](int value) {
                std::cout << value << '\n';
            });
}
```

think-cell

Careful: Composing `std::expected` requires the same E

```
std::expected<std::string, DBError> query_value(Key key);
std::expected<int, ParseError> parse_value(std::string const& value);
```

```
auto process_value(Key key)
    -> std::expected<void, std::variant<DBError, ParseError>>
{
    using result_type = std::expected<std::string, std::variant<…>>;
    return result_type(query_value(key))
          .and_then(&parse_value)
          .transform([](int value) {
              std::cout << value << '\n';
          });
}
```

Wishlist: Variadic expected<T, E...>

```
template <typename T, typename ... E>
struct expected;
```

Either holds a `T` (value) or a variant of `E...` (errors); `and_then` automatically accumulates new errors.

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:

think-cell

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:
  - `.transform` to process a new value

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:
  - `.transform` to process a new value
  - `.transform_error` to handle an error

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:
  - `.transform` to process a new value
  - `.transform_error` to handle an error
  - `.and_then` to chain with another optional/failible function

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:
  - `.transform` to process a new value
  - `.transform_error` to handle an error
  - `.and_then` to chain with another optional/failible function
- Unfortunately: mixing different kinds of failure/optionalness is difficult

think-cell

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:
  - `.transform` to process a new value
  - `.transform_error` to handle an error
  - `.and_then` to chain with another optional/failible function
- Unfortunately: mixing different kinds of failure/optionalness is difficult

think-cell

General takeaways

- Use `std::optional` and `std::expected` to model optional and failible results.
- Know your compositions:
  - `.transform` to process a new value
  - `.transform_error` to handle an error
  - `.and_then` to chain with another optional/failible function
- Unfortunately: mixing different kinds of failure/optionalness is difficult

Composing using `.transform`/`.and_then` minimizes use of `if`.

think-cell

Composing I/O

Functional programming is pure

Principle: All functions are pure.

- No side-effects during computation.
- Always result in the same value when called with the same input.

Functional programming is pure

Principle: All functions are pure.

- No side-effects during computation.
- Always result in the same value when called with the same input.

That means:

- No global state.
- No functions that interact with the outside world.

Functional programming is pure

So how do you do *anything*?

think-cell

So how do you do *anything*?

```
int read_int();
void write_int(int i);
```

```
void read_and_write_square() {
    auto i = read_int();
    write_int(i * i);
}
```

think-cell

Idea: Manipulate list of actions

```
io_action<int>  read_int();        // pure
io_action<void> write_int(int i); // pure
```

Idea: Manipulate list of actions

```
io_action<int>  read_int();        // pure
io_action<void> write_int(int i); // pure
```

```
void read_and_write_square() { // pure
    auto i = read_int();
    write_int(i * i); // error: i isn't an int!
}
```

think-cell

Idea: Manipulate list of actions

```
io_action<int>  read_int();        // pure
io_action<void> write_int(int i); // pure
```

```
auto read_and_write_square() { // pure
    return read_int()                                     // io_action<int>
        .transform([](int i) { return i * i; })           // io_action<int>
        .and_then(write_int);                             // io_action<void>
}
```

Idea: Manipulate list of actions

```
io_action<int>  read_int();       // pure
io_action<void> write_int(int i); // pure
```

```
auto read_and_write_square() { // pure
    return read_int()                                  // io_action<int>
        .transform([](int i) { return i * i; })        // io_action<int>
        .and_then(write_int);                          // io_action<void>
}
```

```
int main() {
    auto action = read_and_write_square();
    action.run(); // non-pure
}
```

think-cell

Pure functions in C++

Functions in C++ don't need to be pure.

Functions in C++ don't need to be pure.

But it's still a good idea to separate I/O from computation:

- The computation takes the input, produces the output without doing any I/O directly.
- Much easier to test.
- Much easier to swap out input/output procedures.

think-cell

Composing actions can still be useful.

First declaratively compose actions, then execute later.

Actions for multi-threading

```
void do_something(std::execution::scheduler auto sched) {
    auto a = std::execution::schedule(sched)
        | std::execution::then([] {
            std::cout << fib(10) << '\n';
        });
    auto b = std::execution::schedule(sched)
        | std::execution::then([] {
            std::cout << is_prime(42) << '\n';
        });
    std::this_thread::sync_wait(
        std::execution::when_all(a, b)
    );
}
```

General takeways

- Declarative build plans, execute them later.
- Separate I/O from computation.

think-cell

General takeways

- Declarative build plans, execute them later.
- Separate I/O from computation.

Composing actions minimizes impure functions.

Conclusion

That seemed repetitive

- A range contains zero or more values
  - `stdv::transform` changes those values
  - `stdv::transform_join` changes those values using a function that returns a range
  - `stdv::join` flattens nested ranges

That seemed repetitive

- A range contains zero or more values
  - `stdv::transform` changes those values
  - `stdv::transform_join` changes those values using a function that returns a range
  - `stdv::join` flattens nested ranges
- A `std::optional`/`std::expected` contain a value or nothing/an error
  - `.transform` changes the value
  - `.and_then` changes the value by another failible function
  - We can write `join` that flattens nested `std::optional`/`std::expected`

think-cell

That seemed repetitive

- A range contains zero or more values
  - `stdv::transform` changes those values
  - `stdv::transform_join` changes those values using a function that returns a range
  - `stdv::join` flattens nested ranges
- A `std::optional`/`std::expected` contain a value or nothing/an error
  - `.transform` changes the value
  - `.and_then` changes the value by another failible function
  - We can write `join` that flattens nested `std::optional`/`std::expected`
- `io_action` contains a value in the future
  - `.transform` changes the value
  - `.and_then` changes the value using another I/O action
  - We can write `join` that flattens nested `io_action`

think-cell

That seemed repetitive

- A range contains zero or more values
  - `stdv::transform` changes those values
  - `stdv::transform_join` changes those values using a function that returns a range
  - `stdv::join` flattens nested ranges
- A `std::optional`/`std::expected` contain a value or nothing/an error
  - `.transform` changes the value
  - `.and_then` changes the value by another failible function
  - We can write `join` that flattens nested `std::optional`/`std::expected`
- `io_action` contains a value in the future
  - `.transform` changes the value
  - `.and_then` changes the value using another I/O action
  - We can write `join` that flattens nested `io_action`

Those are all monads.

think-cell

Monad

C++ Definition

A monad is a type that implements the operations

- `.transform`
- `.and_then`
- `.join`

together with some way to build a monad from a value.

think-cell

Monad

C++ Definition

A monad is a type that implements the operations

- `.transform`
- `.and_then`
- `.join`

together with some way to build a monad from a value.

Monads enable composition.

think-cell

think-cell

Join our team as a
C++ Developer
or Intern

think-cell.com/career