

Functional Programming in C++

Jonathan Müller — @foonathan | think-cell C++ Meetup | 2025-11-05

Over two decades of experience building a market-leading C++ add-in for Microsoft Office

- think-cell is the gold standard for creating professional PowerPoint presentations.
- Founded in 2002, we now have 1,200,000+ users at over 30,000 companies.
- Our software is built entirely in C++ and fully integrated into Microsoft Office.
- We are a member of Standard C++ Foundation, run the Berlin C++ Meetup and talk at conferences.
- We are driven by excellence, setting ambitious goals and fostering the growth of exceptional talent.

We're hiring: think-cell.com/career/dev

What is functional programming?

Definition

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

Definition

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

- C and C++

Definition

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

- C and C++
- CPU

Definition

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

- C and C++
- CPU
- IKEA manual

Definition

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

Definition

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

- Haskell, Prolog

Definition

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

- Haskell, Prolog
- formal grammar

Definition

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

- Haskell, Prolog
- formal grammar
- thermostat

Definition

Declarative programming by composing functions.

Definition

Declarative programming by composing functions.

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

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
```

Why functional programming?

Why functional programming?

Functional programming eliminates state.

Functional programming eliminates state.

- No bugs due to invalid states, broken invariants.

Functional programming eliminates state.

- No bugs due to invalid states, broken invariants.
- No bugs due to complex intertwined object references.

Functional programming eliminates state.

- No bugs due to invalid states, broken invariants.
- No bugs due to complex intertwined object references.
- No bugs due to race conditions, easier to parallelize.

Why not functional programming?

Why not functional programming?

Functional programming does not map neatly to hardware.

Functional programming does not map neatly to hardware.

- Harder to optimize.

Functional programming does not map neatly to hardware.

- Harder to optimize.
- Harder to use native tools (debuggers, profilers).

Functional programming does not map neatly to hardware.

- Harder to optimize.
- Harder to use native tools (debuggers, profilers).
- Harder to interact with native libraries and OS APIs.

Should you use functional programming?

Using a functional programming language is a trade-off.

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 is all about COMPOSITION.

Functional programming is all about COMPOSITION.

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

1. Composing algorithms.
2. Composing functions.
3. Composing I/O.

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

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

```
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.

Example: Biggest odd magnitude

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

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.

Problem: Eager composition

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

Solution: Lazy composition

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

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

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

Example: Biggest odd magnitude

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

Example: Find sum of two integers

Problem statement

Input Two lists of integers, a number target

Output Two integers, one from each list, whose sum is target

Example: Find sum of two integers

Problem statement

Input Two lists of integers, a number target

Output Two integers, one from each list, whose sum is target

Input [1, 2, 3, 4], [5, 6, 7, 8], 12

Output (4, 8)

Example: Find sum of two integers

```
auto find_sum(auto&& rng1, auto&& rng2, int target)
-> std::optional<std::tuple<int, int>>
{
    for (int a : rng1) {
        for (int b : rng2) {
            if (a + b == target) {
                return std::make_tuple(a, b);
            }
        }
    }
    return std::nullopt;
}
```

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 target.

Example: Find sum of two integers

```
auto find_sum(auto&& rng1, auto&& rng2, int target)
-> std::optional<std::tuple<int, int>>
{
    // 1. Generate all pairs.
    auto combinations = std::cartesian_product(rng1, rng2);
    // 2. Find the pair with the sum `target`.
    auto iter = std::find_if(
        combinations,
        [target](auto p) {
            return std::get<0>(p) + std::get<1>(p) == target;
        }
    );
    return iter == combinations.end() ? std::nullopt : *iter;
}
```

Example: Find sum of two integers

```
auto find_sum(auto&& rng1, auto&& rng2, int target)
-> std::optional<std::tuple<int, int>>
{
    // 1. Generate all pairs.
    auto combinations = std::cartesian_product(rng1, rng2);
    // 2. Find the pair with the sum `x`.
    return tc::find_first_if<tc::return_value_or_none>(
        combinations,
        [target](auto a, auto b) {
            return a + b == target;
        }
    );
}
```

Example: Find sum of two integers

```
auto find_sum(auto&& rng1, auto&& rng2, int target)
-> std::optional<std::tuple<int, int>>
{
    // 1. Generate all pairs.
    auto combinations = std::cartesian_product(rng1, rng2);
    // 2. Find the pair with the sum `target`.
    auto iter = std::find_if(
        combinations,
        [target](auto p) {
            return std::get<0>(p) + std::get<1>(p) == target;
        }
    );
    return iter == combinations.end() ? std::nullopt : *iter;
}
```

Aside: Make it parallel

```
auto find_sum(auto&& rng1, auto&& rng2, int target)
```

C++26

```
-> std::optional<std::tuple<int, int>>
{
    // 1. Generate all pairs.
    auto combinations = std::cartesian_product(rng1, rng2);
    // 2. Find the pair with the sum `target` - in parallel!
    auto iter = std::find_if(
        std::execution::par_unseq,
        combinations,
        [target](auto p) {
            return std::get<0>(p) + std::get<1>(p) == target;
        }
    );
    return iter == combinations.end() ? std::nullopt : *iter;
}
```

Aside: Making it efficient

```
auto find_sum(auto&& rng1, auto&& rng2, int target)
-> std::optional<std::tuple<int, int>>
{
    // Create an efficient lookup structure for `rng1`.
    auto lookup = rng1 | stdr::to<std::unordered_set<int>>;
    // Find a pair by using the lookup structure.
    auto iter = stdr::find_if(
        rng2,
        [target, &lookup](int b) {
            return lookup.contains(target - b);
        }
    );
    return iter == rng2.end()
        ? std::nullopt : std::make_tuple(*iter, target - *iter);
}
```

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

```
std::size_t longest_increasing_subsequence(auto&& rng)
{
    std::size_t max_length = 0, current_length = 0;
    int previous = INT_MIN;
    for (int x : rng) {
        if (x > previous) { // add to current subsequence
            ++current_length;
        } else { // start a new subsequence
            max_length = std::max(max_length, current_length);
            current_length = 1;
        }
        previous = x;
    }
    return std::max(max_length, current_length);
}
```

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.

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]
    );
}
```

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

```
std::string reverse_words(std::string str)
{
    char previous = ' ';
    for (auto iter = str.begin(); iter != str.end(); ) {
        if (std::isalpha(*iter) && !std::isalpha(previous)) { // new word
            auto start = iter;
            while (iter != str.end() && std::isalpha(*iter))
                ++iter;
            std::reverse(start, iter);
        } else {
            ++iter;
        }
        previous = iter[-1];
    }
    return str;
}
```

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

```
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;
            else
                return chunk;
        }) // ["olleH", " ", "dlroW", "!"]
        | stdv::join | stdr::to<std::string>(); // "olleH dlroW!"
}
```

Example: Reverse words

```
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

```
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!"
}
```

- Try to identify small building blocks for solving a problem
 - 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*`

- Try to identify small building blocks for solving a problem
 - 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`.

Composing functions

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

```
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<T>
```

Either holds a T or is empty.

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

Either holds a T or is empty.

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

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

```
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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
void unsubscribe_from_mailing_list(UserID id) {  
    lookup_user(id)                // User or nothing  
    .transform(&User::email)       // Email or nothing  
    .transform([&](Email const& e) { // action  
        subscriber_list.remove(e);  
        return std::monostate{};  
    });  
}
```

What if not every user has an email address?

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

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);  
    }  
}
```

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

```
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;
}
```

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

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

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

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

```
void unsubscribe_from_mailing_list(UserID id) {  
    lookup_user(id)                // User or nothing  
    .and_then(&User::email)        // Email or nothing  
    .transform([&](Email const& e) { // action  
        subscriber_list.remove(e);  
        return std::monostate{};  
    });  
}
```

What if a user can have multiple email addresses?

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

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 or nothing  
        .transform(&User::email);         // Emails or nothing  
  
    stdr::for_each(  
        stdv::join(opt_emails),           // Emails  
        [&](Email const& e) {            // action  
            subscriber_list.remove(e);  
        }  
    );  
}
```

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 or nothing  
        | stdv::transform(&User::email) | stdv::join, // Emails or nothing  
        [&](Email const& e) {                          // action  
            subscriber_list.remove(e);  
        }  
    );  
}
```

Composing optional functions with ranges

C++29?

`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 or nothing  
        | stdv::transform_join(&User::email), // Emails or nothing  
        [&](Email const& e) {           // action  
            subscriber_list.remove(e);  
        }  
    );  
}
```

```
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); });  
    }  
}
```

```
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<T, E>
```

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

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

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

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

```
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); });  
    }  
}
```

```
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

```
// 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::unexpected, this->error());  
}
```

```
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); });  
    }  
}
```

```
// 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::unexpected, f(this->error()));  
}
```

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

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);  
    }  
}
```

Flatten a nested expected

```
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());
}
```

```
// 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());  
}
```

```
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 if success  
            subscriber_list.remove(e);  
        })  
        .transform_error([&](DBError const& e) { // action if error  
            log_error(e);  
            retry_operation_later([id] { unsubscribe_from_mailing_list(id); });  
            return std::monostate{}; // explicit swallow  
        });  
}
```

```
void unsubscribe_from_mailing_list(UserId id) {  
    std::expected<void, DBError> result = ...;  
    std::expected<void, std::monostate> handled  
        = result.transform_error([&](DBError const& e) {  
            log_error(e);  
            retry_operation_later([id] { unsubscribe_from_mailing_list(id); });  
            return std::monostate{}; // explicit swallow  
        });  
    assert(!handled);  
}
```

```
void unsubscribe_from_mailing_list(UserId id) {  
    std::expected<void, DBError> result = ...;  
    std::expected<void, std::expected<void, std::monostate>> handled  
        = result.transform_error([&](DBError const& e) {  
            log_error(e);  
            retry_operation_later([id] { unsubscribe_from_mailing_list(id); });  
            return std::expected<void, std::monostate>{}; // explicit success  
        });  
    assert(!handled);  
}
```

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

```
void unsubscribe_from_mailing_list(UserId id) {  
    std::expected<void, DBError> result = ...;  
    std::expected<void, std::monostate> handled  
        = result.or_else([&](DBError const& e) {  
        log_error(e);  
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });  
        return std::expected<void, std::monostate>{}; // explicit success  
    });  
    assert(!handled);  
}
```

```
void unsubscribe_from_mailing_list(UserId id) {  
    std::expected<void, DBError> result = ...;  
    std::expected<void, std::monostate> handled  
        = result.or_else([&](DBError const& e) {  
        log_error(e);  
        retry_operation_later([id] { unsubscribe_from_mailing_list(id); });  
        return std::expected<void, noreturn_t>{}; // explicit success  
    });  
    assert(!handled);  
}
```

std::monostate ("top type")

- Single value

noreturn_t ("bottom type")

- Zero values

Proposed by P3549.

std::monostate ("top type")

- Single value
- tuple</* empty */>

noreturn_t ("bottom type")

- Zero values
- variant</* empty */>

Proposed by P3549.

std::monostate ("top type")

- Single value
- `tuple</* empty */>`
- Logically return type of `void f() {}`

noreturn_t ("bottom type")

- Zero values
- `variant</* empty */>`
- Logically return type of `abort()`

Proposed by P3549.

std::monostate ("top type")

- Single value
- `tuple</* empty */>`
- Logically return type of `void f() {}`
- `expected<T, monostate>` is isomorphic to `optional<T>`

noreturn_t ("bottom type")

- Zero values
- `variant</* empty */>`
- Logically return type of `abort()`
- `expected<T, noreturn_t>` is isomorphic to `T`

Proposed by P3549.

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

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>>  
{  
    return query_value(key)  
        .and_then(&parse_value) // doesn't compile  
        .transform([](int value) {  
            std::cout << value << '\n';  
        });  
}
```

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.

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

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

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

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.

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*?

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);  
}
```

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!
}
```

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
}
```

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.

Composing actions can still be useful.

First declaratively compose actions, then execute later.

C++26

```
namespace stdx = std::execution;
```

```
stdx::sender auto async_read_int();  
stdx::sender auto async_write_int(int i);
```

```
stdx::sender auto read_and_write_square() {  
    return async_read_int()  
        | stdx::then([](int i) { return i * i; }) // "transform"  
        | stdx::let_value(async_write_int);      // "and_then"  
}
```

```
int main() {  
    auto action = read_and_write_square();  
    std::this_thread::sync_wait(action); // "run"  
}
```

C++26

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

```
stdx::sender auto async_read_int();  
stdx::sender auto async_write_int(int i);
```

```
stdx::sender auto read_and_write_fib(stdx::scheduler auto sched) {  
    return stdx::schedule(sched)  
        | stdx::let_value(async_read_int)  
        | stdx::then([](int i) {  
            return fib(i);  
        })  
        | stdx::let_value(async_write_int);  
}
```

General takeaways

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

- 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
 - `std::transform` changes those values
 - `std::join` flattens nested ranges
 - We can write a `transform_join` changes those values using a function that returns a range

That seemed repetitive

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

That seemed repetitive

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

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

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.

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.

A monad is a “box” of some value that can be transformed while keeping it inside the box.

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.

A monad is a “box” of some value that can be transformed while keeping it inside the box.

Monads enable composition.

Functional programming is all about COMPOSITION.

Monads enable composition.

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

Feedback and new speakers wanted!

