Unix-Style Composable Operations in C++

Ankur M. Satle

https://ankursatle.wordpress.com | /ankursatle

ankursatle@gmail.com | @ankursatle

$whoami

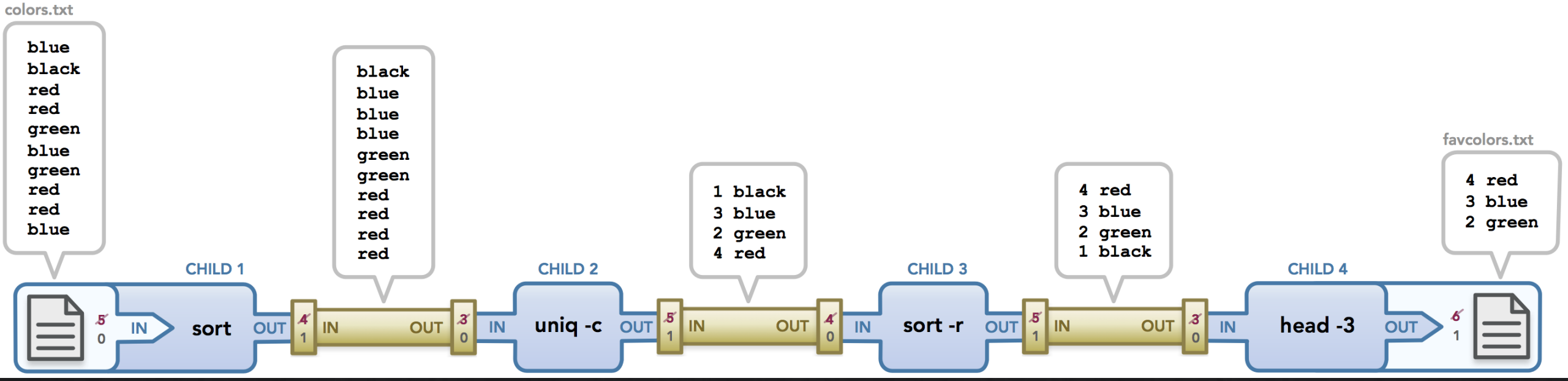
- Architect at EXFO
- Love C++
- High-Performance Implementations
- Cloud Native products
- Neophilia

- https://ankursatle.wordpress.com
- ankursatle@gmail.com
- https://www.linkedin.com/in/ankursatle/
- https://twitter.com/AnkurSatle

Unix piped commands – top 3 colors

- Very easy to compose

```
$ sort colors.txt | uniq -c | sort -r | head -3 > favcolors.txt
```



What this talk is about

- Ways to realize | chained | operations
  - Just like Unix commands
- Generic chaining implementation
- Error handling
- Customization
- Flexible, clean ways of code with visible clutter or boiler-plate
  - Implement the boiler-plate once and not turn back!

This talk uses but is NOT purely about

- We use the following concepts
  - Functional Programming & Monad
  - <ranges>
  - <concepts>
  - More…
- The focus though remains on chaining operations

- Stay away from the not-so-good parts! The slides have slideware code…

The same command operations in C++

- Problem: output top 3 colours
- Use <algorithms>
- Do not compose well
- Many intermediate steps

```cpp
int main(int, char**) {
    //ifstream in_colors("colors.txt");

    map<string, int> counts;
    for_each(istream_iterator<string>{cin}, istream_iterator<string>{},
        [&counts](string s) { counts[std::move(s)]++; }
    );

    vector<pair<int, string>> freq;
    for (const auto& [color, count] : counts) {
        freq.push_back(make_pair(count, color));
    }
    auto last_interested = freq.size() > 3? begin(freq) + 3: end(freq);
    partial_sort(begin(freq), last_interested, end(freq), greater{});

    for_each (begin(freq), last_interested, [](const auto& p) {
        cout << p.first << ' ' << p.second << '\n';
    });
}
```

- https://ankursatle.godbolt.org/z/W4sMYd3hG

Squares of nos. divisible by 3

- Build a vector of squares from only the elements in another vector that are divisible by 3

```
std::vector<int> input = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
std::vector<int> intermediate, output;

std::copy_if(input.begin(), input.end(), std::back_inserter(intermediate),
   [](const int i) { return i%3 == 0; });
std::transform(intermediate.begin(), intermediate.end(), std::back_inserter(output),
   [](const int i) {return i*i; });
```

- Source: https://docs.microsoft.com/en-us/cpp/standard-library/ranges?view=msvc-170

Employing Ranges to do the same!

```
std::vector<int> input =  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto divisible_by_three = [](const int n) {return n % 3 == 0; };
auto square = [](const int n) {return n * n; };

auto x = input  | std::views::filter(divisible_by_three)
                | std::views::transform(square);

for (int i : x) {
    std::cout << i << '\n';
}
```

Ranges are impressively expressive

- Clean and straight-forward
- Extendible
- Convey intent
- Efficient!

What If

- When not working with ranges, containers
- Can core application logic/business logic be so clear?

- Of course, the answer was…

- **Why not!**

A Typical Queue Connect

- Application connects to Kafka/MQ
- Creation of the connection is a multi-step process
- Start-up configuration is in file
- Dynamically instantiate instance
- Connect
- Subscribe to events

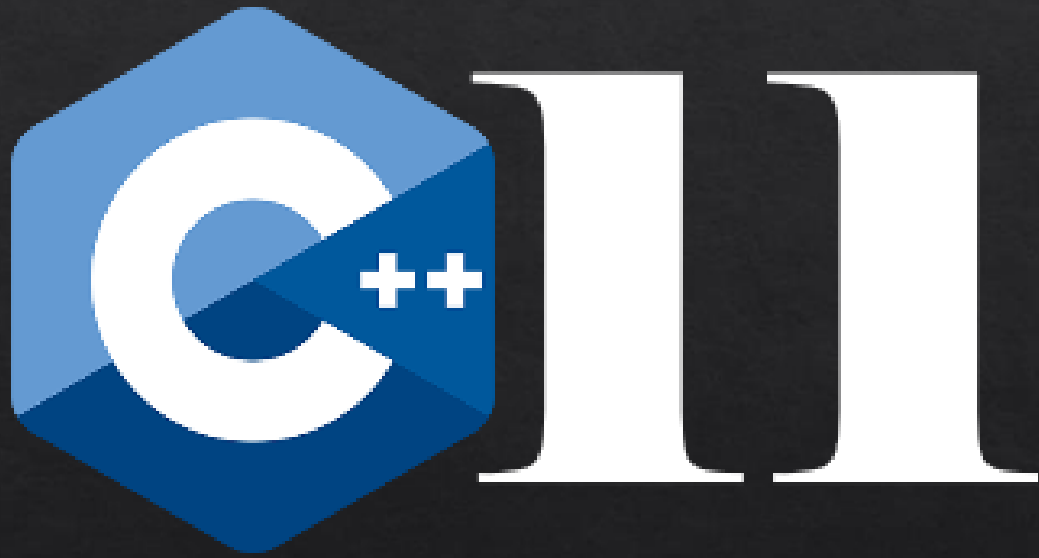
Queue Connect

```
if (auto fname = get_env("MQInitClientConfigFile")) {
    auto contents = get_file_contents(fname);
    if (!contents) { /*error - file I/O*/ throw bad_config; }
    auto client = make_client(contents);
    if (client) {
        if (client->apply_config() != true &&
            client->connect() != true &&
            client->subscribe() != true) {
                //error - Queue config/connect/subscription error
        }
    } else {    //error - client creation
    }
} else {    //error - environment config not found
}
```

Desired code

- Drum roll…

```
return get_env("MQInitClientConfigFile")
        | get_file_contents
        | make_client
        | apply_config
        | connect
        | subscribe;
```



How?

- Let's start simple. With

```
string get_file_contents(const std::string& fname) { ... };
```

- We can implement

```
auto operator|(string&& s, string (*fn)(string)) -> string {
    return fn(std::move(s));
}
```

- Now, this is possible

```
get_env("MQInitClientConfigFile") | get_file_contents;
```

The Callable need not be a function!

- C++ has many callables

```
template<typename T, typename Callable>
auto operator|(T&& t, Callable&& fn) -> typename std::result_of<Callable(T)>::type
{
    return std::forward<Callable>(fn)(std::forward<T>(t));
}
```

- Now

```
auto get_file_contents = [](const std::string& fname) { ... };
get_env("MQInitClientConfigFile") | get_file_contents;
```

Sometimes, member functions may be called

- Write a free function instead!

`connect(client)` instead of `client->connect()`

- If not possible, write a wrapper

```
Client* connect(Client* client) {
    client->connect();
    return client;
}
```

- Or use available functionality, if the return type is suitable for further chaining

```
make_client | std::mem_fn(&Client::connect)
```

To Chain Operations

- Callables take what the previous steps gives
- Return a new value
- Chaining is possible only if the ins & outs of subsequent operations match

- With that:

```
return get_env("MQInitClientConfigFile")
        | get_file_contents
        | make_client
        | apply_config
        | connect
        | subscribe;
```

Errors & Clean-up

- Dynamically allocated instance if worked upon by many functions
- Possibility of memory leak?
- Use unique_ptr

```
std::unique_ptr<Client> connect(std::unique_ptr<Client> client) {
    if (client->connect()) {
        return client;
    } else {
        throw connect_error{};   //error – client clean-up guaranteed
    }
}
```

Errors & Clean-up

- Unique exception types help track the step that failed

```
try {
    get_env("MQInitClientConfigFile")
        | get_file_contents
        | make_client
        | apply_config
        | connect
        | subscribe;
} catch(ex1) {
    return nullopt;
} catch(ex2) { ...
}
```

I see this everywhere now…

- An application relaying events from one source to another (with or without a value add)
- Some examples

```
rx_msg | parse | validate | extract | store
rx_msg | parse | validate | extract | enrich | send
rx_http_req | extract_body | validate | actionize | make_response | send
trigger | make_msg | encode<json> | encrypt(certificate) | send
```

Sy Brand
@TartanLlama

Monadic extensions for std::optional

Adds .transform, .and_then, and .or_else to std::optional

This was my paper! 🥰🥰🥰

Old version

```
::optional<image> get_cute_cat (const image& im
 auto cropped = crop_to_cat(img);
 if (!cropped) {
   return std::nullopt;
 }
 auto with_tie = add_bow_tie(*cropped);
 if (!with_tie) {
   return std::nullopt;
 }
 auto with_sparkles = make_eyes_sparkle(*with_t
 if (!with_sparkles) {
   return std::nullopt;
 }
 return add_rainbow(make_smaller(*with_sparkles
```

ALT

```
sion

nal<image> get_cute_cat (const im
 crop_to_cat(img)
 .and_then(add_bow_tie)
 .and_then(make_eyes_sparkle)
 .transform(make_smaller)
 .transform(add_rainbow);
```

ALT

9:24 pm · 21 Mar 2022 · Twitter Web App

16 Retweets **8** Quote Tweets **292** Likes

C++23 - Evolution!

```cpp
// Old version

std::optional<image> get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
      return std::nullopt;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
      return std::nullopt;
    }
    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
      return std::nullopt;
    }
    return add_rainbow(make_smaller(*with_sparkles));
}
```

```cpp
// New version

std::optional<image> get_cute_cat (const image& img) {
    return crop_to_cat(img)
            .and_then(add_bow_tie)
            .and_then(make_eyes_sparkle)
            .transform(make_smaller)
            .transform(add_rainbow);
}
```

Around the corners?

- No mechanism to know which step failed
- No error handling
- Use the mechanisms discussed in this talk

What it could be!..

```
std::optional<image> get_cute_cat(const image& img) {
    return img  | crop_to_cat
                | add_bow_tie
                | make_eyes_sparkle
                | make_smaller
                | add_rainbow;
}
```

This is how… on one slide – The operators

```cpp
//operator| instead of std::invoke at the start of a chain
template<typename T, typename Callable>
requires std::invocable<Callable, const T&> || std::invocable<Callable, T&>
    || std::invocable<Callable, T&&>
auto operator|(T&& t, Callable&& fn) {
    return std::invoke(std::forward<Callable>(fn), std::forward<T>(t));
}

//operator| instead of std::optional<T>::and_then() to chain operations
template<Optional Opt, typename Callable>
requires std::invocable<Callable, typename Opt::value_type&&>
    && Optional<std::invoke_result_t<Callable, typename Opt::value_type&&>>
auto operator|(Opt&& opt, Callable&& fn) {
    return opt.has_value()? std::invoke(std::forward<Callable>(fn),
            std::forward<typename Opt::value_type&&>(*std::move(opt))):
        std::nullopt;
}

//operator| instead of std::optional<T>::transform() to chain transformations
template<Optional Opt, typename Callable>
requires std::invocable<Callable, typename Opt::value_type&&>
    && (!Optional<std::invoke_result_t<Callable, typename Opt::value_type&&>>)
auto operator|(Opt&& opt, Callable&& fn)
    -> std::optional<std::invoke_result_t<Callable, typename Opt::value_type&&>> {
    return opt.has_value()? std::make_optional(std::invoke(std::forward<Callable>(fn),
            std::forward<typename Opt::value_type&&>(*std::move(opt)))):
        std::nullopt;
}

//operator| instead of std::optional<T>::or_else() to chain operations
template<Optional Opt, typename Callable>
requires std::invocable<Callable>
    && std::convertible_to<std::invoke_result_t<Callable>, typename Opt::value_type>
typename Opt::value_type operator|(Opt&& opt, Callable&& fn) {
    return opt.has_value()? *std::move(opt): std::invoke(std::forward<Callable>(fn));
}
```

operator| for Callable(T)

```cpp
//operator| instead of std::invoke at the start of a chain
template<typename T, typename Callable>
requires std::invocable<Callable, const T&> || std::invocable<Callable, T&>
    || std::invocable<Callable, T&&>
auto operator|(T&& t, Callable&& fn) {
    return std::invoke(std::forward<Callable>(fn), std::forward<T>(t));
}
```

and_then

- operator| for `(optional<T>, U(T)) -> optional<U>`

```
//operator| instead of std::optional<T>::and_then() to chain operations
template<Optional Opt, typename Callable>
requires std::invocable<Callable, typename Opt::value_type&&>
    && Optional<std::invoke_result_t<Callable, typename Opt::value_type&&>>
auto operator|(Opt&& opt, Callable&& fn) {
    return opt.has_value()? std::invoke(std::forward<Callable>(fn),
            std::forward<typename Opt::value_type&&>(*std::move(opt))):
        std::nullopt;
}
```

transform

- operator| for `(optional<T>, optional<U>(T)) -> optional<U>`

```
//operator| instead of std::optional<T>::transform() to chain transformations
template<Optional Opt, typename Callable>
requires std::invocable<Callable, typename Opt::value_type&&>
    && (!Optional<std::invoke_result_t<Callable, typename Opt::value_type&&>>)
auto operator|(Opt&& opt, Callable&& fn)
    -> std::optional<std::invoke_result_t<Callable, typename Opt::value_type&&>> {
    return opt.has_value()? std::make_optional(std::invoke(std::forward<Callable>(fn),
            std::forward<typename Opt::value_type&&>(*std::move(opt)))):
        std::nullopt;
}
```

or_else

- Returns the optional itself if it contains a value. Otherwise, result of `Callable()`
- Below needs correction to return Opt and not Opt::value_type

```
//operator| instead of std::optional<T>::or_else() to chain operations
template<Optional Opt, typename Callable>
requires std::invocable<Callable>
    && std::convertible_to<std::invoke_result_t<Callable>, typename Opt::value_type>
typename Opt::value_type operator|(Opt&& opt, Callable&& fn) {
    return opt.has_value()? *std::move(opt): std::invoke(std::forward<Callable>(fn));
}
```

A concept for optional

- To leverage subsumption for overload resolution

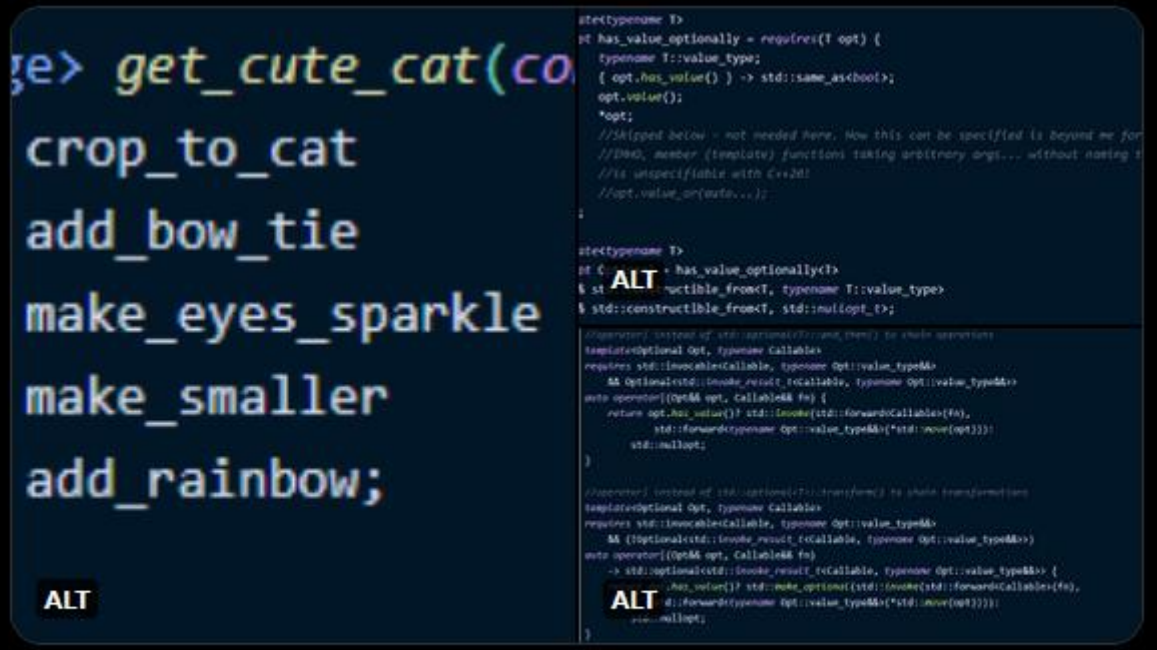
```
template<typename T>
concept has_value_optionally = requires(T opt) {
        typename T::value_type;
        { opt.has_value() } -> std::same_as<bool>;
        opt.value();
        *opt;
        //Skipped below - not needed here. How this can be specified is beyond me for now!
        //IMHO, member (template) functions taking arbitrary args... without naming them
        //is unspecifiable with C++20!
        //opt.value_or(auto...);
    };

template<typename T>
concept Optional = has_value_optionally<T>
    && std::constructible_from<T, typename T::value_type>
    && std::constructible_from<T, std::nullopt_t>;
```

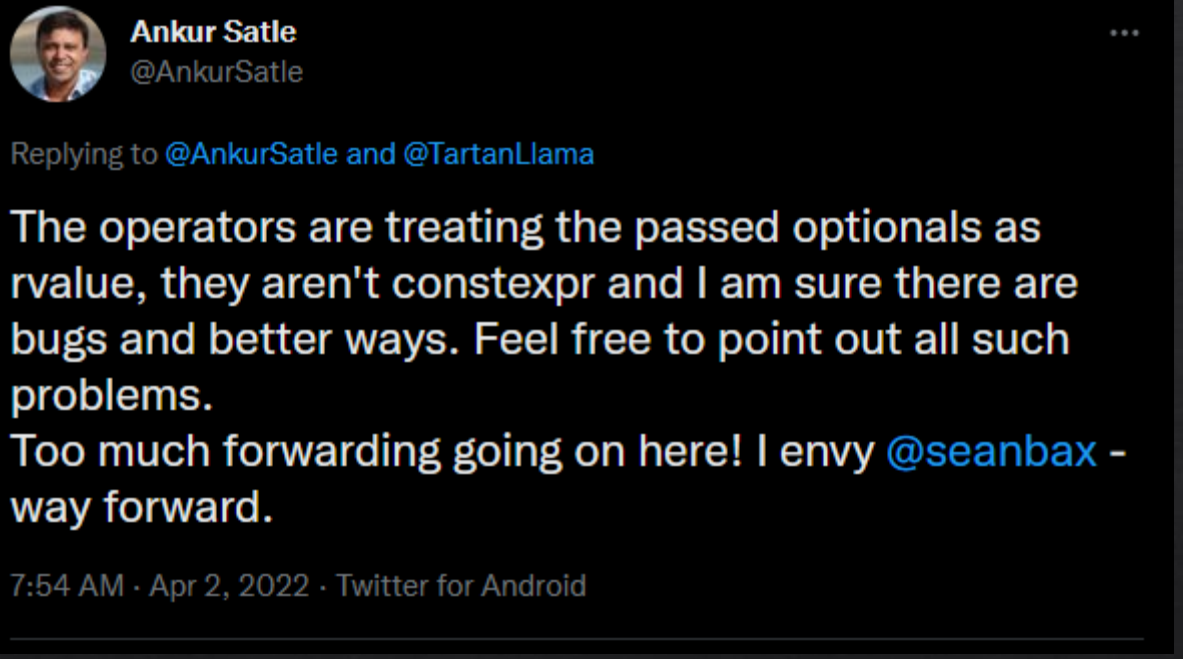
Ankur Satle
@AnkurSatle

Replying to @TartanLlama

C++23 code IS concise!
Having to write and_then & transform was still verbose. I thought, some invisible code could help us here... I defined operator| and a concept. Pipes for the win! Not std quality, perhaps PoC quality. Critique welcome on my try.
See: godbolt.org/z/hnsWT895W



6:47 AM · Apr 2, 2022 · Twitter Web App


Ankur Satle
@AnkurSatle
Replying to @AnkurSatle and @TartanLlama
The operators are treating the passed optionals as rvalue, they aren't constexpr and I am sure there are bugs and better ways. Feel free to point out all such problems.
Too much forwarding going on here! I envy @seanbax - way forward.
7:54 AM · Apr 2, 2022 · Twitter for Android

Chaining Operations

- Get functional!
- Make Pure Composable functions
- Use value semantics

- May not fit every case
  - Branching – pass the same optional to multiple functions or conditionally call one of two fns
  - Asynchrony
  - Multiple returns from a function
- Creating the right concepts may be hard
  - But you have this problem of abstraction anyway!
- Understand the style and have fun
  - Some details may not be obvious

I do not have time to make all the mistakes myself

Please share your learning…

ankursatle@gmail.com

समाप्त

धन्यवाद