Functional Composable Operations with Unix-Style Pipes in C++

ANKUR SATLE

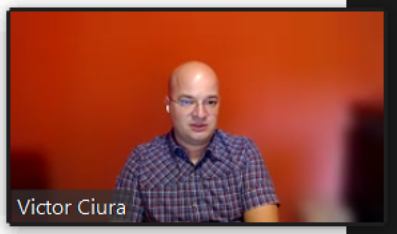
Functional, Composable Operations with Unix-Style | Pipes | in | C++

Ankur M. Satle

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

ankursatle@gmail.com |  @ankursatle

It's all about | pipelines



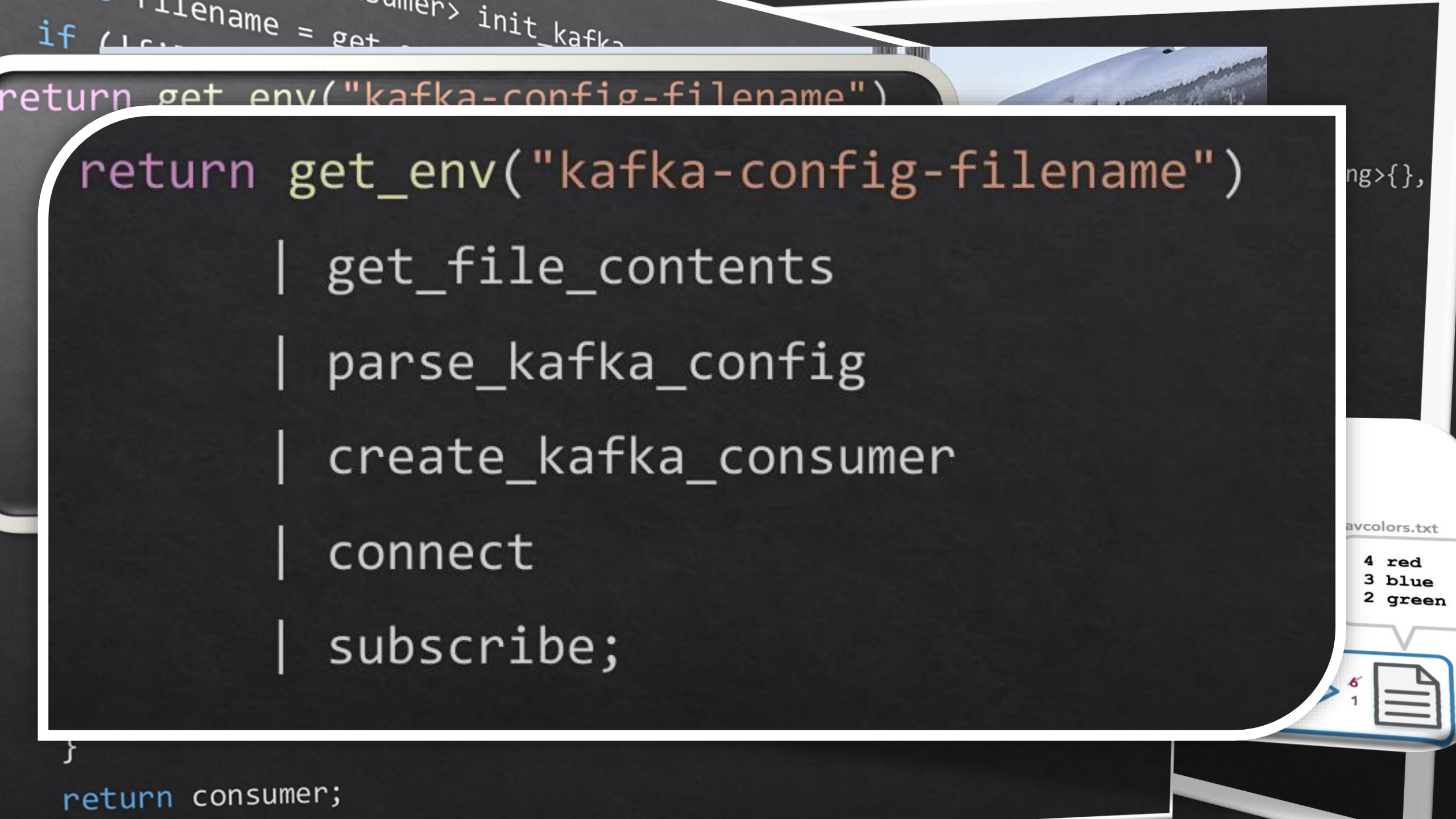
Taking inspiration from Doug McIlroy's UNIX shell script,

write an algorithm in your favorite programming language,

that solves the same problem: word frequencies

The previous talk by Victor Ciura

```
return get_env("kafka-config-filename")
    | get_file_contents
    | parse_kafka_config
    | create_kafka_consumer
    | connect
    | subscribe;
```



$whoami

- Architect at EXFO
- Polyglot when it comes to Natural languages
- But my mother-tongue is C++
- High-Performance Products
- Cloud Native products
- CppIndia 

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

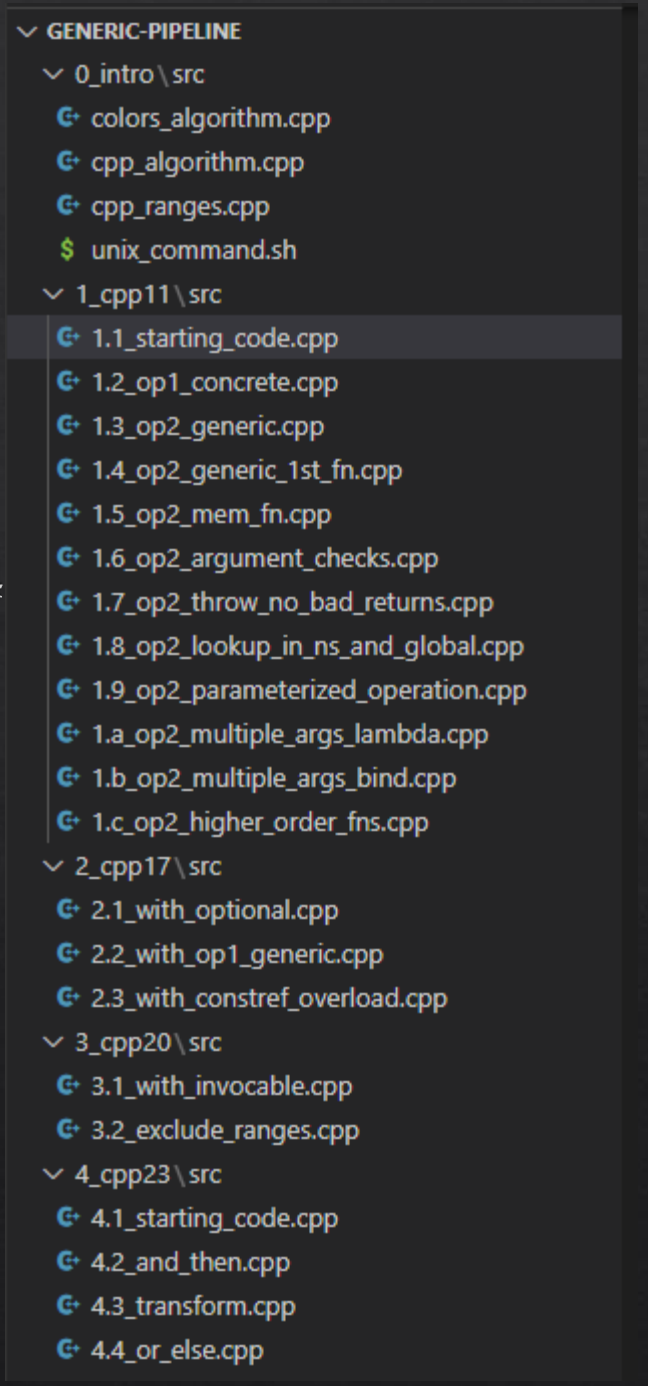
Functional Composable Operations

- Ways to realize | chained | operations
  - Just like Unix commands
- C++11 Generic chaining implementation
  - Error handling
  - Customization
  - Resource management
- C++17 discipline with std::optional
- Ensure how not to interfere with C++20 ranges pipelines using concepts
- Discuss C++23 monadic interface & how we could pipe to have the same effect
- Flexible, clean ways of code with visible clutter or boiler-plate
  - Implement the boiler-plate once and not turn back!

Code available on GitHub

- Check my repo for the implementation & to follow along
- All-in-one header: inc/generic_pipeline.hpp
- https://github.com/sankurm/generic-pipeline
- Slides have code from these files available in sequential order: 1.1 to 4.4

https://github.com/sankurm/generic-pipeline



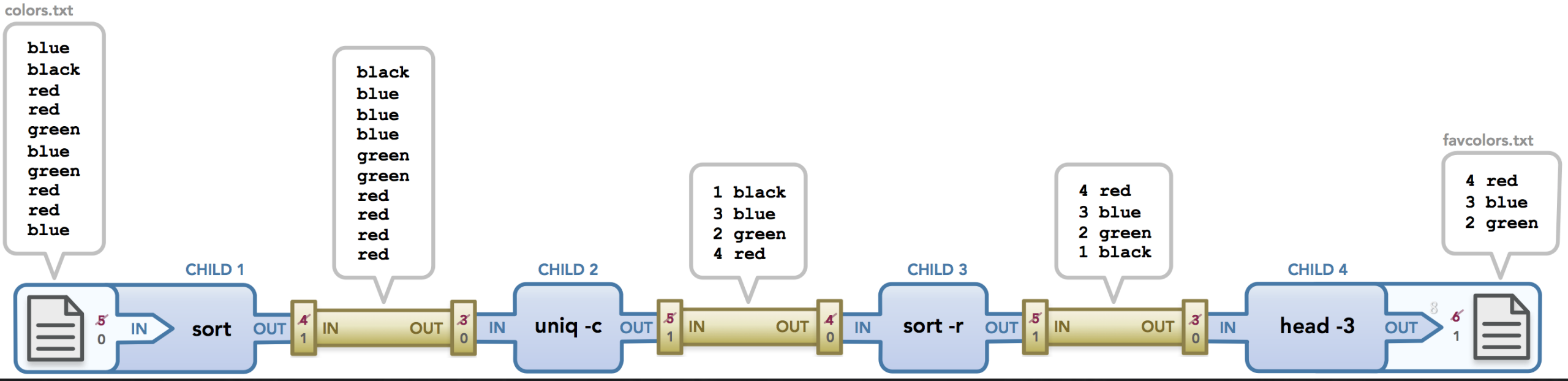
Unix piped commands – top 3 colors

- Very easy to compose

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

```
$ grep "warning: " compile.log | sed -e 's/^.*warning: //1' | uniq | sort | uniq
```

```
$ git log --since=1.day | grep '^Author:' | awk '{print $3}' | xargs -n1 mail -s "Build Failure Alert: Urgent action needed"
```



The same command operations in C++

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

```
int main(int, char**) {
    auto& in_colors = std::cin;
    map<string, int> counts;
    for_each(istream_iterator<string>{in_colors}, istream_iterator<string>{},
        [&counts](string s) { counts[std::move(s)]++; }
    );

    using cnt_clr = pair<int, string>;
    vector<cnt_clr> freq;
    std::transform(begin(counts), end(counts), std::back_inserter(freq),
        [](const pair<string, int>& p){ return std::make_pair(p.second,
p.first); });

    auto last = freq.size() > 3? begin(freq) + 3: end(freq);
    partial_sort(begin(freq), last, end(freq), greater<cnt_clr>{});

    for_each (begin(freq), last, [](const cnt_clr& 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!

I’m impressed!

I see pipes helping everywhere!

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

Consumer Initialization

```
kafka_consumer init_kafka() {
    std::string fname = get_env("kafka-config-filename");
    if (fname.empty()) { throw env_error{}; }

    auto contents = get_file_contents(std::move(fname));
    if (contents.empty()) { throw file_error{}; }

    auto config = parse_kafka_config(std::move(contents));
    if (!config) { throw json_error{}; }

    auto consumer = create_kafka_consumer(std::move(config));
    if (!consumer) { throw creation_error{}; }

    if (!consumer.connect()) { throw connect_error{}; }
    if (!consumer.subscribe()) { throw subscribe_error{}; }


    return consumer;
}
```

We are building this

- The destination

```
struct kafka_consumer
{
    kafka_consumer(const kafka_config& config) {}
    kafka_consumer(kafka_config&& config) {}

    bool connect() { return true; }
    bool subscribe() { return true; }

    operator bool() { return true; }
};
```

Using these

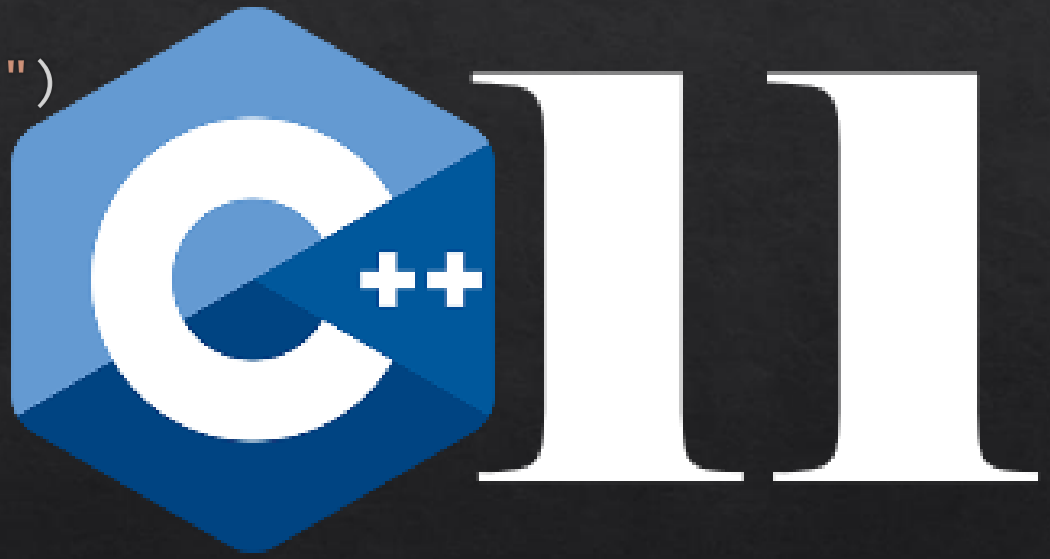
- The path

```
std::string get_env(std::string&& varname);
std::string get_file_contents(std::string&& filename);
kafka_config parse_kafka_config(std::string&& config);
kafka_consumer create_kafka_consumer(kafka_config&& config);
bool kafka_consumer::connect();
bool kafka_consumer::subscribe();
```

Desired code

- Drum roll…

```
kafka_consumer init_kafka() {
    return get_env("kafka-config-filename")
           | get_file_contents
           | parse_kafka_config
           | create_kafka_consumer
           | connect
           | subscribe;
}
```



How?

- Let's start simple. With

```
std::string get_file_contents(std::string&& filename) { ... }
```

- We can implement

```
using Callable = std::string(std::string&&);
auto operator|(std::string&& val, Callable fn) -> std::string {
    return fn(std::move(val));
}
```

- Now, this is possible

```
auto contents = get_env("kafka-config-filename") | get_file_contents;
if (contents.empty()) { throw file_error{}; }
```

The Callable need not be a function!

- C++ has many callables taking any type

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

- Now

```
auto contents = get_env("kafka-config-filename") | get_file_contents;
if (contents.empty()) { throw file_error{}; }
```

Generic!

- In fact, the generic implementation allowed us to write this too:

```
auto config = get_env("kafka-config-filename")
                | get_file_contents
                | parse_kafka_config;
if (!config) { throw json_error{}; }
```

- Beware – type conversion cannot take place when invoking templates
- The T & the passed parameter of Callable _MUST_ match

Member functions may be called

- Write a free function instead!

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

- If not possible, write a wrapper or a lambda

```
kafka_consumer connect(kafka_consumer&& consumer)
{ consumer.connect(); return consumer; }
```

- Or

```
[](kafka_consumer&& consumer) { consumer.subscribe(); return consumer; };
```

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

```
create_kafka_consumer | connect | std::mem_fn(&kafka_consumer::subscribe);
```

We are not checking returned values now!

- We could check the arguments passed to functions

```
kafka_consumer create_kafka_consumer(kafka_config&& config) {
    if (!config) { throw json_error{}; }
    return kafka_consumer{std::move(config)};
}
kafka_consumer connect(kafka_consumer&& consumer) {
    if (!consumer) { throw creation_error{}; }
    consumer.connect();
    return consumer;
}
```

- That is exactly what not to do!
- It bakes in assumption about the previous step

Raise error, where encountered

- Throw where the error occurs

```
kafka_config parse_kafka_config(std::string&& config) {
    if (/*parse config ==*/ false) { throw json_error{}; }
    return kafka_config{};
}
kafka_consumer create_kafka_consumer(kafka_config&& config) {
    return kafka_consumer{std::move(config)}; //constructor would throw
}
kafka_consumer connect(kafka_consumer&& consumer) {
    if (!consumer.connect()) { throw connect_error{}; }
    return consumer;
}
```

Our Chain of Operations is now ready!

- With that, the following is a reality!

```
kafka_consumer init_kafka() {
    return get_env("kafka-config-filename")
            | get_file_contents
            | parse_kafka_config
            | create_kafka_consumer
            | connect
            | subscribe;
}
```

Variations

Operator is in global namespace!

- I tried to move the operator to a generic namespace

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

- But the operator was not visible to any namespace other than framework ☹
- Operator| needs to be global!

No Namespace Please!

- On compilation

```
$ g++ -std=c++11 -Wall -Werror ../1_*/src/1.8_*.cpp -o ./1.8.out
../1_cpp11/src/1.8_op2_lookup_in_ns_and_global.cpp: In function ‘app::kafka_consumer framework1::init_kafka()’:
../1_cpp11/src/1.8_op2_lookup_in_ns_and_global.cpp:83:17: error: no match for ‘operator|’ (operand types are ‘std::string’ {aka ‘std::__cxx11::basic_string
<char>’} and ‘std::string(std::string&&)’ {aka ‘std::__cxx11::basic_string<char>(std::__cxx11::basic_string<char>&&)’})
   82 |         return app::get_env("kafka-config-filename")
      |                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      |                           |
      |                           std::string {aka std::__cxx11::basic_string<char>}
}
   83 |                | app::get_file_contents
      |                ^ ~~~~~~~~~~~~~~~~~~~~~~
      |                  |
{     |                  std::string(std::string&&) {aka std::__cxx11::basic_string<char>(std::__cxx11::basic_string<char>&&)}
In file included from /usr/include/c++/11/ios:42,
                 from /usr/include/c++/11/ostream:38,
                 from /usr/include/c++/11/iostream:39,
                 from ../1_cpp11/src/1.8_op2_lookup_in_ns_and_global.cpp:2:
/usr/include/c++/11/bits/ios_base.h:87:3: note: candidate: ‘constexpr std::_Ios_Fmtflags std::operator|(std::_Ios_Fmtflags, std::_Ios_Fmtflags)’
   87 |   operator|(_Ios_Fmtflags __a, _Ios_Fmtflags __b)
      |   ^~~~~~~~
/usr/include/c++/11/bits/ios_base.h:87:27: note:   no known conversion for argument 1 from ‘std::string’ {aka ‘std::__cxx11::basic_string<char>’} to ‘std::
_Ios_Fmtflags’
```

Calling function templates

- A function needed to be in the chain may be templated

```
enum config_type { json, xml, yaml, config_map };

template<config_type format>
kafka_config parse_kafka_config(std::string&& config) {
    if (/* parsing as per format == */ false) { throw json_error{}; }
    return kafka_config{};
}
```

Templated functions in the chain

- For templated functions, use them with the parameter

```
kafka_consumer init_kafka() {
    return get_env("kafka-config-filename")
           | get_file_contents
           | parse_kafka_config<xml>
           | create_kafka_consumer
           | connect
           | subscribe;
}
```

Functions taking multiple args

- If another data is needed for a function like this:

```
kafka_consumer connect(kafka_consumer&& consumer, const certificate&
cert) {
    if (!consumer.connect(cert)) { throw connect_error{}; }
    return consumer;
}
```

Function taking multiple args

```
a_basic_cpp11_op2_function_with_2_args.cpp: In function '{anonymous}::kafka_consumer {anonymous}::init_kafka()':
a_basic_cpp11_op2_function_with_2_args.cpp:79:25: error: no match for 'operator|' (operand types are 'std::__success_type<{anonymous}::kafka_consumer>::type' {aka '{anonymous}::kafka_consumer'} and '{anonymous}::kafka_consumer({anonymous}::kafka_consumer&&, const {anonymous}::certificate&)')
   75 |         return get_env("kafka-config-filename")
      |                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   76 |                         | get_file_contents
      |                         ~~~~~~~~~~~~~~~~~~~
   77 |                         | parse_kafka_config<xml>
      |                         ~~~~~~~~~~~~~~~~~~~~~~~~~
   78 |                         | create_kafka_consumer
      |                         ~~~~~~~~~~~~~~~~~~~~~~~
      |                         |
      |                         std::__success_type<{anonymous}::kafka_consumer>::type {aka {anonymous}::kafka_consumer}
   79 |                         | connect
      |                         ^ ~~~~~~~~
      |                           |
      |                           {anonymous}::kafka_consumer({anonymous}::kafka_consumer&&, const {anonymous}::certificate&)
a_basic_cpp11_op2_function_with_2_args.cpp:10:6: note: candidate: 'template<class T, class Callable> typename std::result_of<Callable(T)>::type operator|(T&&, Callable&&)'
   10 | auto operator|(T&& val, Callable&& fn) -> typename std::result_of<Callable(T)>::type {
      |      ^~~~~~~~
a_basic_cpp11_op2_function_with_2_args.cpp:10:6: note:   template argument deduction/substitution failed:
a_basic_cpp11_op2_function_with_2_args.cpp: In substitution of 'template<class T, class Callable> typename std::result_of<Callable(T)>::type operator|(T&&, Callable&&) [with T = {anonymous}::kafka_consumer; Callable = {anonymous}::kafka_consumer (&)({anonymous}::kafka_consumer&&, const {anonymous}::certificate&)]':
a_basic_cpp11_op2_function_with_2_args.cpp:79:27:   required from here
a_basic_cpp11_op2_function_with_2_args.cpp:10:6: error: no type named 'type' in 'struct std::result_of<{anonymous}::kafka_consumer (&({anonymous}::kafka_consumer))({anonymous}::kafka_consumer&&, const {anonymous}::certificate&)>'
-bash: ./a_basic_cpp11_op2_function_with_2_args: No such file or directory
```

Function taking multiple args - lambda

- Lambdas just work!

```
kafka_consumer init_kafka() {
    auto cert = get_certificate();
    return get_env("kafka-config-filename")
            | get_file_contents
            | parse_kafka_config<xml>
            | create_kafka_consumer
            //This needs C++14 :(
            //| [cert = get_certificate()](kafka_consumer&& consumer)
                { return connect(std::move(consumer), cert); }
            | [&cert](kafka_consumer&& consumer)
              { return connect(std::move(consumer), cert); }
            | subscribe;
}
```

Function taking multiple args – std::bind

- Standard library can help

```
kafka_consumer init_kafka() {
    return get_env("kafka-config-filename")
            | get_file_contents
            | parse_kafka_config<xml>
            | create_kafka_consumer
            //| connect
            | std::bind(connect, std::placeholders::_1, get_certificate())
            | subscribe;
}
```

- Works like a charm!

Higher-order functions

- Higher order functions can combine multiple functions
- Any of the functions in the call could be coming from a higher-level function
- As far as it can process the passed type

Higher-order functions

- In a message processing chain like this

- We may want to look-up in a data-structure and let it process the message

```
auto process(Msg&& msg, Callable1 lookup_call, Callable2 handle_msg_for_call)
{
    auto call = lookup_call(msg);
    if (!call) { throw call_not_found{}; }
    return handle_msg_for_call(call, std::move(msg));
}
```

Functional Programming

- Operations are pure functions
  - Depend only on arguments
  - Have no side-effects, just return or throw
- Work with values
  - Work with objects with ownership
- Return a new object created or the original object passed
  - `buffer<byte> serialize(Message)`
  - `T add_timestamp(T)`
  - Enables composability – keep the chain going

Errors & Clean-up

- Memory leak for dynamically allocated instance passed around?
- Use unique_ptr

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

Errors & Clean-up

- Unique exception types help track the step that failed

```
kafka_consumer init_kafka() {
    try {
        return get_env("kafka-config-filename")
                | get_file_contents
                | parse_kafka_config<xml>
                | create_kafka_consumer
                | connect
                | subscribe;
    } catch (creation_error) {
        throw;
    }
}
```

Passing the first parameter via operator|

- Possible but a matter of taste – ranges allow this

```
kafka_consumer init_kafka() {
    //using namespace std::string_literals;
    //Can use "kafka-config-filename"s as they need C++14
    return std::string("kafka-config-filename")
            | get_env
            | get_file_contents
            | parse_kafka_config<xml>
            | create_kafka_consumer
            | connect
            | subscribe;
}
```

Pipes can be used everywhere

- 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
```

Uniformity with C++17 std::optional

std::optional<T> to stop the chain

- std::optional<T> can signal lack of an object & that a step failed
- Good enough to stop the chain if you didn't want to know which step failed

```
std::optional<kafka_consumer> create_kafka_consumer(kafka_config&&
config) {
    return std::make_optional<kafka_consumer>(std::move(config));
}
```

- No need to use std::unique_ptr
- Stay with values!

```
auto cons = create_kafka_consumer(std::move(config.value()));
if (!cons) { return std::nullopt; }

auto consumer = cons.value();
```

std::optional<T>

- Raison d'etre: std::optional also helps with there not being any special nullptr-like value that signifies non-existence of the object
- This is very desirable for many applications
- Eliminates the need to check before use of domain objects
- No interference with Domain objects due to constructs

Queue Connect – C++17


```
//Without pipes
std::optional<kafka_consumer> init_kafka() {
    auto fname = get_env("kafka-config-filename");
    if (!fname) { return std::nullopt; }

    auto contents = get_file_contents(std::move(fname.value()));
    if (!contents) { return std::nullopt; }

    auto config = parse_kafka_config(std::move(contents.value()));
    if (!config) { return std::nullopt; }

    auto cons = create_kafka_consumer(std::move(config.value()));
    if (!cons) { return std::nullopt; }

    auto consumer = cons.value();
    if (consumer.connect() && consumer.subscribe()) {
        return cons;
    }
    return std::nullopt;
}
```

Pipe to unwrap std::optional

- New overload

```
//Handle unwrapping of std::optional
template<typename T, typename Callable>
auto operator|(std::optional<T>&& opt, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return opt?
        std::invoke(std::forward<Callable>(fn), *std::move(opt)):
        std::nullopt;
}
```

Pipes work with std::optional

- Instantly, we are in the land of pipes!

```
std::optional<kafka_consumer> init_kafka() {
    using namespace std::string_literals;
    return "kafka-config-filename"s
            | get_env
            | get_file_contents
            | parse_kafka_config
            | create_kafka_consumer
            | connect
            | subscribe;
}
```

One overload for const ref std::optional

- Dealing with a const std::optional<T>&

```
template<typename T, typename Callable>
auto operator|(const std::optional<T>& opt, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return opt?
        std::invoke(std::forward<Callable>(fn), *opt): std::nullopt;
}
```

Possible to work with lvalues

```
std::optional<kafka_consumer> init_kafka() {
    using namespace std::string_literals;
    auto opt_consumer = "kafka-config-filename"s
                      | get_env
                      | get_file_contents
                      | parse_kafka_config
                      | create_kafka_consumer;
    return opt_consumer | connect | subscribe;
}
```

Refinement with C++20

concept and ranges magic

Constraining the operator|

- The operator as defined so far works on any types
- The second parameter _MUST_ be a Callable
- Let's state the intent

```
template<typename T, typename Callable>
requires std::invocable<Callable, T>
auto operator|(T&& val, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return std::invoke(std::forward<Callable>(fn), std::forward<T>(val));
}
```

- Operator is now considered by overload resolution only with the correct callable

Interference with std::ranges

- With our operator, when we write some ranges code:

```
std::vector ints{1, 3, 5, 7, 9};
auto pipeline = ints | std::views::transform([](int n) { return n*n; });
for (auto n : pipeline) { std::cout << n << ' '; }
```

- We have disrupted std::ranges code

```
$ g++ -std=c++20 -Wall -Werror ../3_*/src/3.2_*.cpp -o ./3.2.out
../3_cpp20/src/3.2_exclude_ranges.cpp: In function ‘int main(int, char**)’:
../3_cpp20/src/3.2_exclude_ranges.cpp:86:26: error: ambiguous overload for ‘operator|’ (operand types are ‘std::vector<int, std::allocator<int> >’ and ‘std
::ranges::views::__adaptor::_Partial<std::ranges::views::_Transform, main(int, char**)::<lambda(int)> >’)
   86 |     auto pipeline = ints | std::views::transform([](int n) { return n*n; });
      |                     ~~~~ ^ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      |                          |
      |                          std::ranges::views::__adaptor::_Partial<std::ranges::views::_Transform, main(int, char**)::<lambda(
int)> >
      |                     std::vector<int, std::allocator<int> >
../3_cpp20/src/3.2_exclude_ranges.cpp:11:6: note: candidate: ‘std::invoke_result_t<Callable, T> operator|(T&&, Callable&&) [with T = std::vector<int, std::
allocator<int> >&; Callable = std::ranges::views::__adaptor::_Partial<std::ranges::views::_Transform, main(int, char**)::<lambda(int)> >; std::invoke_resul
t_t<Callable, T> = std::ranges::transform_view<std::ranges::ref_view<std::vector<int, std::allocator<int> > >, main(int, char**)::<lambda(int)> >]’
   11 | auto operator|(T&& val, Callable&& fn) -> typename std::invoke_result_t<Callable, T> {
      |      ^~~~~~~~
In file included from ../3_cpp20/src/3.2_exclude_ranges.cpp:7:
/usr/include/c++/11/ranges:782:7: note: candidate: ‘constexpr auto std::ranges::views::__adaptor::operator|(_Range&&, _Self&&) [with _Self = std::ranges::v
iews::__adaptor::_Partial<std::ranges::views::_Transform, main(int, char**)::<lambda(int)> >; _Range = std::vector<int, std::allocator<int> >&]’
  782 |       operator|(_Range&& __r, _Self&& __self)
      |       ^~~~~~~~
```

The operator| must be constrained

- It should not shadow the pipe operator for ranges

```
template<typename T, typename Callable>
requires (not std::ranges::range<T> && std::invocable<Callable, T>)
auto operator|(T&& val, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return
        std::invoke(std::forward<Callable>(fn), std::forward<T>(val));
}
```

std::ranges work!

```
int main(int argc, char** argv) {
    auto cons = init_kafka();
    if (cons) { std::cout << "Consumer creation successful\n"; }
    else { std::cout << "Consumer creation failed\n"; }

    std::vector ints{1, 3, 5, 7, 9};
    auto pipeline = ints | std::views::transform([](int n) { return n*n;
});
    for (auto n : pipeline) { std::cout << n << ' '; }
    std::cout << '\n';

    return 0;
}
```

Into the future – C++23

Monadic operations on std::optional

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);
}
```


Source: https://wg21.link/p0798

For our use-case

- Our code with C++23 std::optional looks like:

```
return get_env("kafka-config-filename")
        .transform(get_file_contents)
        .transform(parse_kafka_config)
        .transform(create_kafka_consumer)
        .and_then(connect)
        .and_then(subscribe);
```

Feels like we are going back!

- Big plus:
  - member functions do not add to the global namespace
  - operator| must be global
- Big minus: So verbose!

- Can’t our pipes do the equivalent functionality of the monadic operations!?

and_then

std::optional<T>::and_then

```
template< class F >
constexpr auto and_then( F&& f ) &;                (1)  (since C++23)

template< class F >
constexpr auto and_then( F&& f ) const&;           (2)  (since C++23)

template< class F >
constexpr auto and_then( F&& f ) &&;               (3)  (since C++23)

template< class F >
constexpr auto and_then( F&& f ) const&&;          (4)  (since C++23)
```

Returns the result of invocation of f on the contained value if it exists. Otherwise, returns an empty value of the return type.
The return type (see below) must be a specialization of `std::optional`. Otherwise, the program is ill-formed.

- Essentially, this: `(std::optional<T>, std::optional<U>(T)) -> std::optional<U>`
- Need to test that the Callable indeed returns an optional

Return-type of Callable!?

- So far, the return type of the functions was assumed to be std::optional

```
template<typename T, typename Callable>
requires std::invocable<Callable, T>
auto operator|(std::optional<T>&& opt, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return opt?
        std::invoke(std::forward<Callable>(fn), *std::move(opt)):
        std::nullopt;
}
```

Return-type of Callable!?

- We wish to specify & check clearly. But how!?
- Cannot check it as:

```
std::convertible_to<std::invoke_return_t<...>, std::optional<T>>
```

- Don't know the type T for an arbitrary Callable

- No concept in namespace std to check whether a type is an optional
- Need to create one ourselves like this

```
template<typename T>
concept basic_optional = requires { ... }
```

Concept basic_optional

```
template<typename T>
concept basic_optional = requires (T t) {
    typename T::value_type;
    std::convertible_to<T, bool>;
    std::same_as
        <std::remove_cvref<decltype(*t)>, typename T::value_type>;
    std::constructible_from<T, std::nullopt_t>;
};
static_assert(basic_optional<std::optional<int>>);
```

and_then using operator|

```
template<typename T, typename Callable>
requires std::invocable<Callable, T>
    && basic_optional<typename std::invoke_result_t<Callable, T>>
auto operator|(std::optional<T>&& opt, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return opt?
        std::invoke(std::forward<Callable>(fn), *std::move(opt)):
        std::nullopt;
}
```

and_then using operator|

- The const ref overload

```
template<typename T, typename Callable>
requires std::invocable<Callable, T>
    && basic_optional<typename std::invoke_result_t<Callable, T>>
auto operator|(const std::optional<T>& opt, Callable&& fn)
        -> typename std::invoke_result_t<Callable, T> {
    return opt?
        std::invoke(std::forward<Callable>(fn), *opt): std::nullopt;
}
```

and_then using operator |

- C++23 std::optional with generic-pipe for `and_then`:

```
return get_env("kafka-config-filename")
       .transform(get_file_contents)
       .transform(parse_kafka_config)
       .transform(create_kafka_consumer)
       | connect
       | subscribe;
```

transform

std::optional<T>::transform

```
template< class F >
constexpr auto transform( F&& f ) &;                (1)  (since C++23)

template< class F >
constexpr auto transform( F&& f ) const&;           (2)  (since C++23)

template< class F >
constexpr auto transform( F&& f ) &&;               (3)  (since C++23)

template< class F >
constexpr auto transform( F&& f ) const&&;          (4)  (since C++23)
```

Returns an `std::optional` that contains the result of invocation of f on the contained value if `*this` contains a value. Otherwise, returns an empty `std::optional` of such type.

- Essentially, it is this: `(std::optional<T>, U(T)) -> std::optional<U>`

transform with operator |

- New overload

```
template<typename T, typename Callable>
requires std::invocable<Callable, T>
    && (!basic_optional<typename std::invoke_result_t<Callable, T>>)
auto operator|(std::optional<T>&& opt, Callable&& fn)
        -> std::optional<typename std::invoke_result_t<Callable, T>> {
    return opt? std::make_optional(
        std::invoke(std::forward<Callable>(fn), *std::move(opt))):
        std::nullopt;
}
```

- A similar overload for the const ref case of T

Functions can stay with domain types

- We move from

```
std::optional<kafka_config> parse_kafka_config(std::string&& config) {
    return std::make_optional(kafka_config{});
}
```

- To

```
kafka_config parse_kafka_config(std::string&& config) {
    return kafka_config{};
}
```

- No wrapping/boiler-plate needed

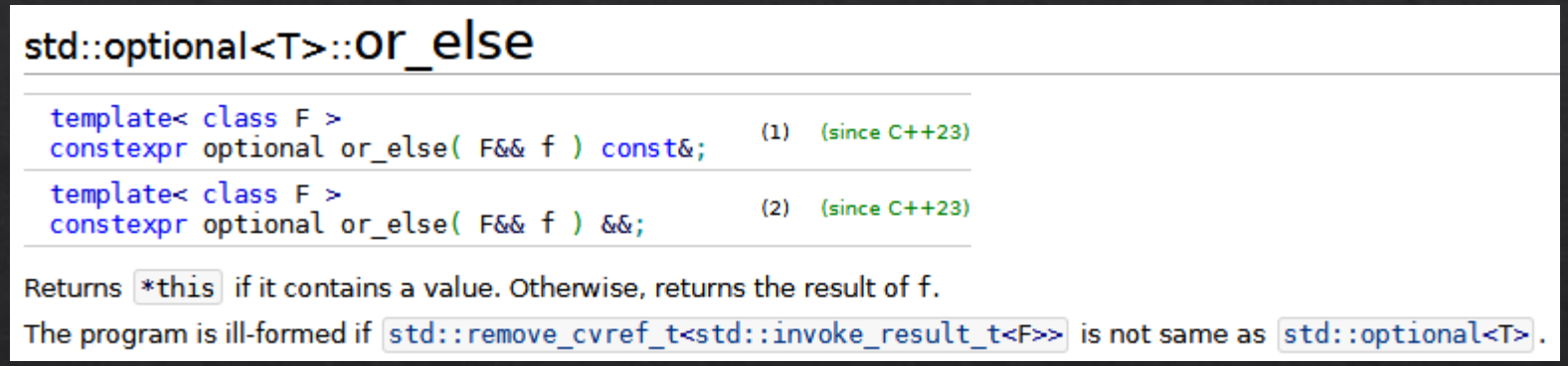
and_then & transform using operator |

- C++23 std::optional with generic-pipe for `and_then` and `transform` :

```
return get_env("kafka-config-filename")
       | get_file_contents
       | parse_kafka_config
       | create_kafka_consumer
       | connect
       | subscribe;
```

or_else

- There is one more function in the std::optional monadic interface



std::optional<T>::or_else

```
template< class F >
constexpr optional or_else( F&& f ) const&;          (1)  (since C++23)

template< class F >
constexpr optional or_else( F&& f ) &&;              (2)  (since C++23)
```

Returns `*this` if it contains a value. Otherwise, returns the result of f.

The program is ill-formed if `std::remove_cvref_t<std::invoke_result_t<F>>` is not same as `std::optional<T>`.

- It is the functional equivalent of the `std::optional<T>::value_or()`
- Essentially this: `(std::optional<T>, std::optional<T>()) -> std::optional<T>`
- We can hack this using pipe but…

Scenario

- One of the functions in the chain may fail
- But we want to proceed with some default action if that happens

```
std::optional<std::string> get_file_contents(std::string&& filename) {
    std::ifstream file(filename, std::ios::in);
    if (!file) { return std::nullopt; }
    return "file-contents-blah-blah";
}
```

- If this was to fail, we would like this default behaviour

```
std::optional<std::string> get_default_config() {
    return "default-contents";
}
```

or_else

- Enabling `or_else` functionality with generic-pipe

```
template<typename T, typename Callable>
requires (!std::invocable<Callable, T>) && std::invocable<Callable> &&
std::same_as<typename std::invoke_result_t<Callable>, std::optional<T>>
auto operator|(std::optional<T>&& opt, Callable&& fn)
        -> typename std::invoke_result_t<Callable> {
    return opt? *opt: std::invoke(std::forward<Callable>(fn));
}
```

- The check `(!std::invocable<Callable, T>)` enforces that this overload is picked only if Callable cannot be invoked with T – this function objects!

and_then, transform & or_else using operator|

- C++23 std::optional with generic-pipe for `and_then`, `transform` & `or_else`:

```
return get_env("kafka-config-filename")
       | get_file_contents | get_default_config
       | parse_kafka_config
       | create_kafka_consumer
       | connect
       | subscribe;
```

- We are striving for expressivity here but looks like we lost some here
- It is not clear what that strange function picking default is going in the chain
- I advise against using generic-pipe for `or_else`

Other wrapping types

- The pipe operator can be extended to other wrapping types
- std::expected – coming in C+23
  - This won't have the monadic interface in C++23
  - But an overload of the pipe operator can surely deal with it!
- std::future
  - This was going to have a different function for continuation – then
  - The operators implemented here are eager
  - Special handling would be needed to make them lazy
  - Maybe the pipe operator should invoke std::future members

Executing operations async

- I have not tried this myself
- There will have to be some way to specify asynchrony

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

Disadvantages

- Operations if too granular, will result in a long chain
  - Cognitive overload & potentially inefficient, even at run-time
- Recurring work – must add specific handling to new types – recurring work
  - E.g. std::future, std::sender, etc.
- operator| may conflict with the generic pipeline operator
  - If the object on RHS implements operator(), the pipe operator kicks in
  - Future or esoteric types
- Due to the long chain, template errors become very long
- Slow overload Resolution

Use Generic Pipeline to your advantage!

- Write simple & clear code with generic-pipeline
- `generic-pipeline` Repository: https://github.com/sankurm/generic-pipeline
- All-in-One header: `generic-pipeline.hpp`

“

Keep making
Effective, Efficient, Elegant
solutions!

”

Ankur M. Satle

Thank you!

References

- cppreference.com
- P0798R8: Monadic operations for std::optional
- P2011R1: A pipeline-rewrite operator
- P0826R0: SFINAE friendly std::bind
- P2387R3: Pipe support for user-defined range adaptors