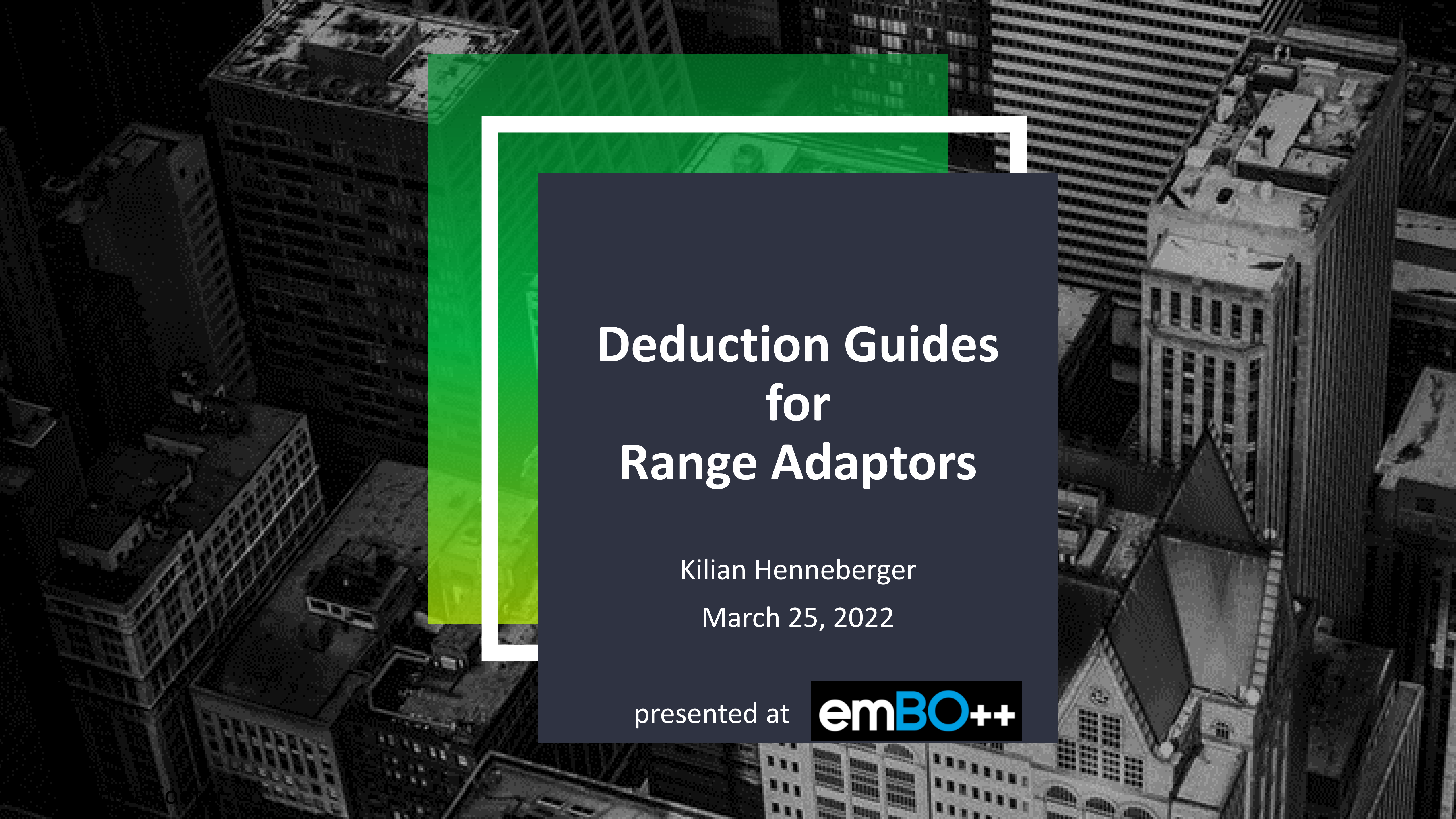


Deduction Guides for Range Adaptors

Kilian Henneberger

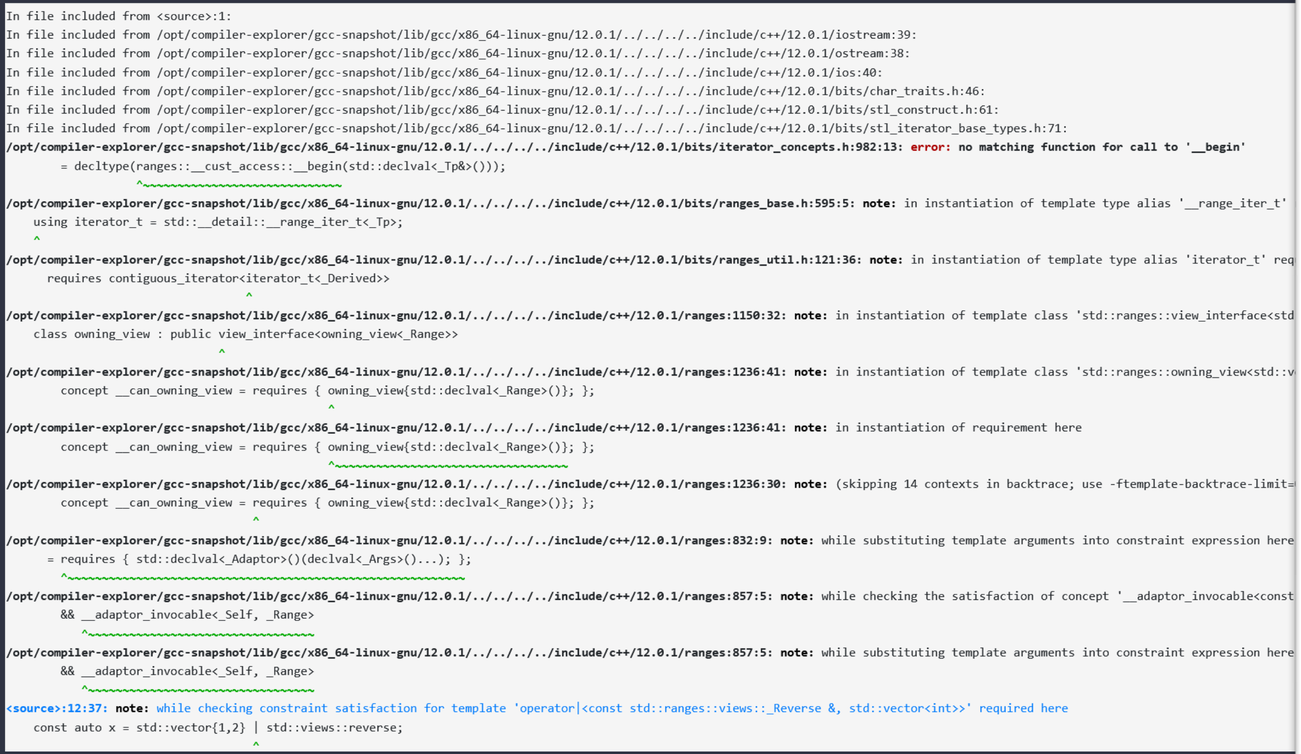
March 25, 2022

presented at emBO++

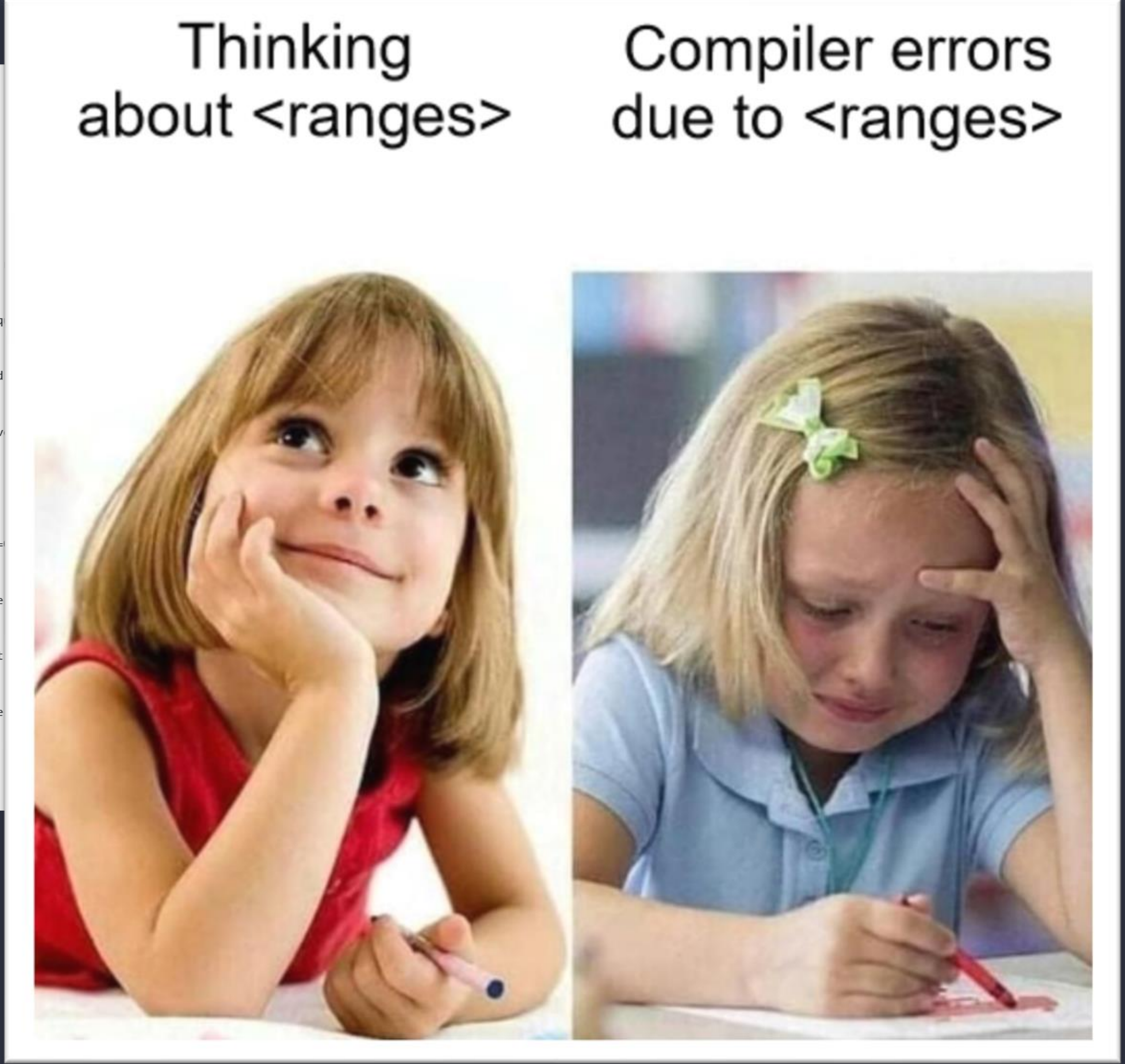
Ranges Library

- C++20 added <ranges>
- Documented on cppreference.com
- Earliest available since
  - GCC libstdc++ 10.1 (May 7, 2020)
  - MSVC STL in VS2019 16.6 (July 15, 2020)
  - Clang libc++ 13.0.0 (Oct 4, 2021)
- There have been several DRs applied to C++20 retrospectively
- C++23 will add further features
- The C++ Standard, cppreference and the STL implementations are not always in sync

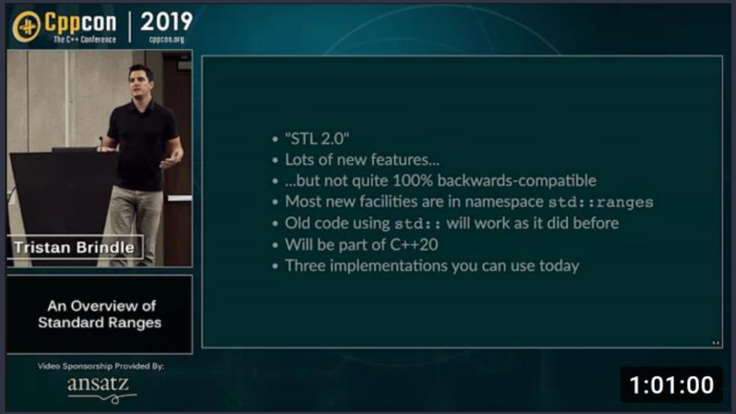
Ranges Library



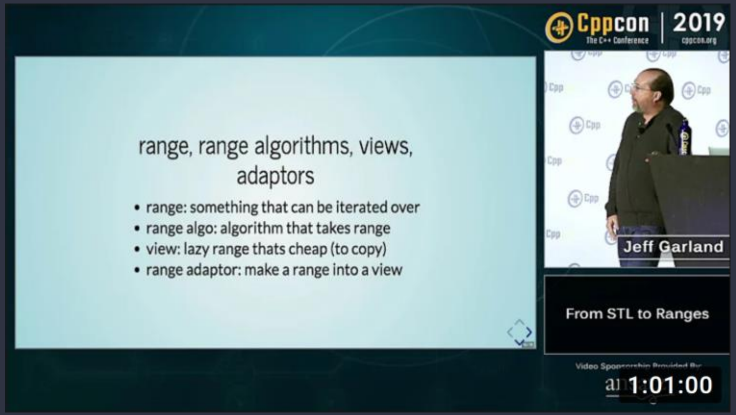
```
In file included from <source>:1:
In file included from /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/iostream:39:
In file included from /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ostream:38:
In file included from /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ios:40:
In file included from /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/bits/char_traits.h:46:
In file included from /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/bits/stl_construct.h:61:
In file included from /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/bits/stl_iterator_base_types.h:71:
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/bits/iterator_concepts.h:982:13: error: no matching function for call to '__begin'
        = decltype(ranges::__cust_access::__begin(std::declval<_Tp&>()));
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/bits/ranges_base.h:595:5: note: in instantiation of template type alias '__range_iter_t'
    using iterator_t = std::__detail::__range_iter_t<_Tp>;
    ^
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/bits/ranges_util.h:121:36: note: in instantiation of template type alias 'iterator_t' req
      requires contiguous_iterator<iterator_t<_Derived>>
                                   ^
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:1150:32: note: in instantiation of template class 'std::ranges::view_interface<std
    class owning_view : public view_interface<owning_view<_Range>>
                               ^
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:1236:41: note: in instantiation of template class 'std::ranges::owning_view<std::v
        concept __can_owning_view = requires { owning_view{std::declval<_Range>()}; };
                                               ^
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:1236:41: note: in instantiation of requirement here
        concept __can_owning_view = requires { owning_view{std::declval<_Range>()}; };
                                               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:1236:30: note: (skipping 14 contexts in backtrace; use -ftemplate-backtrace-limit=
        concept __can_owning_view = requires { owning_view{std::declval<_Range>()}; };
                                    ^
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:832:9: note: while substituting template arguments into constraint expression here
      = requires { std::declval<_Adaptor>()(declval<_Args>()...); };
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:857:5: note: while checking the satisfaction of concept '__adaptor_invocable<const
        && __adaptor_invocable<_Self, _Range>
           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-gnu/12.0.1/../../../../include/c++/12.0.1/ranges:857:5: note: while substituting template arguments into constraint expression here
        && __adaptor_invocable<_Self, _Range>
           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:12:37: note: while checking constraint satisfaction for template 'operator|<const std::ranges::views::_Reverse &, std::vector<int>>' required here
    const auto x = std::vector{1,2} | std::views::reverse;
                                    ^
```



A lot of good talks...



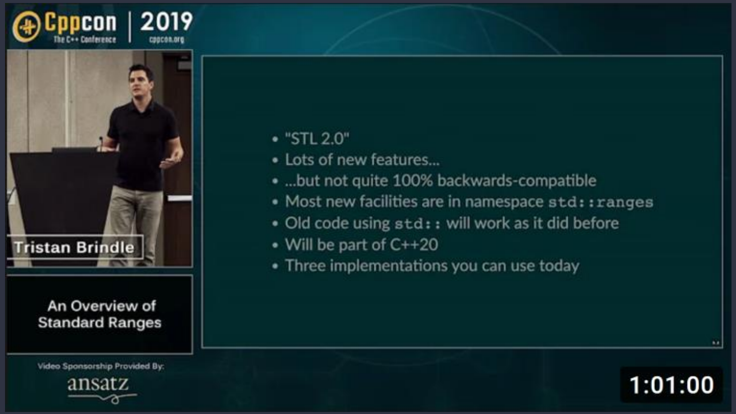
- "STL 2.0"
- Lots of new features...
- ...but not quite 100% backwards-compatible
- Most new facilities are in namespace `std::ranges`
- Old code using `std::` will work as it did before
- Will be part of C++20
- Three implementations you can use today



range, range algorithms, views, adaptors

- range: something that can be iterated over
- range algo: algorithm that takes range
- view: lazy range thats cheap (to copy)
- range adaptor: make a range into a view

A lot of good talks...



- "STL 2.0"
- Lots of new features...
- ...but not quite 100% backwards-compatible
- Most new facilities are in namespace `std::ranges`
- Old code using `std::` will work as it did before
- Will be part of C++20
- Three implementations you can use today

Tristan Brindle

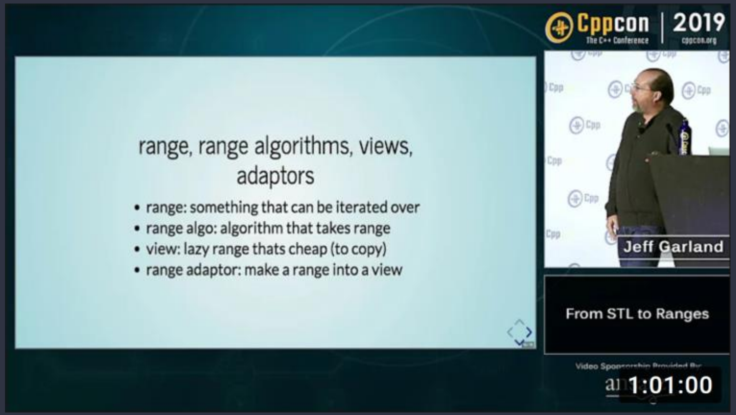
An Overview of Standard Ranges

C++20 Ranges in Practice

Tristan Brindle

Conquering C++20 Ranges

TRISTAN BRINDLE

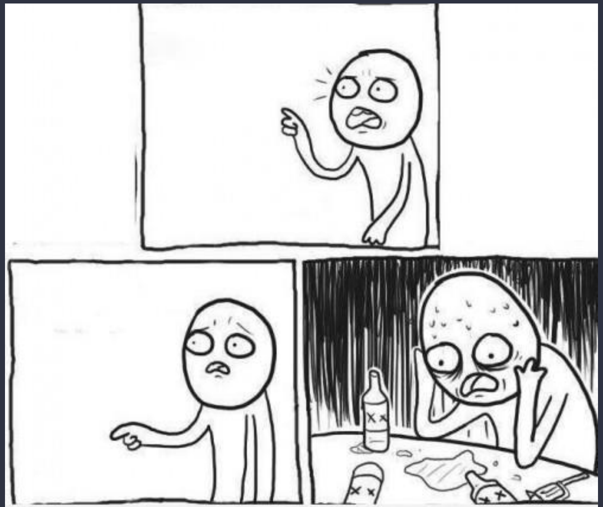


range, range algorithms, views, adaptors

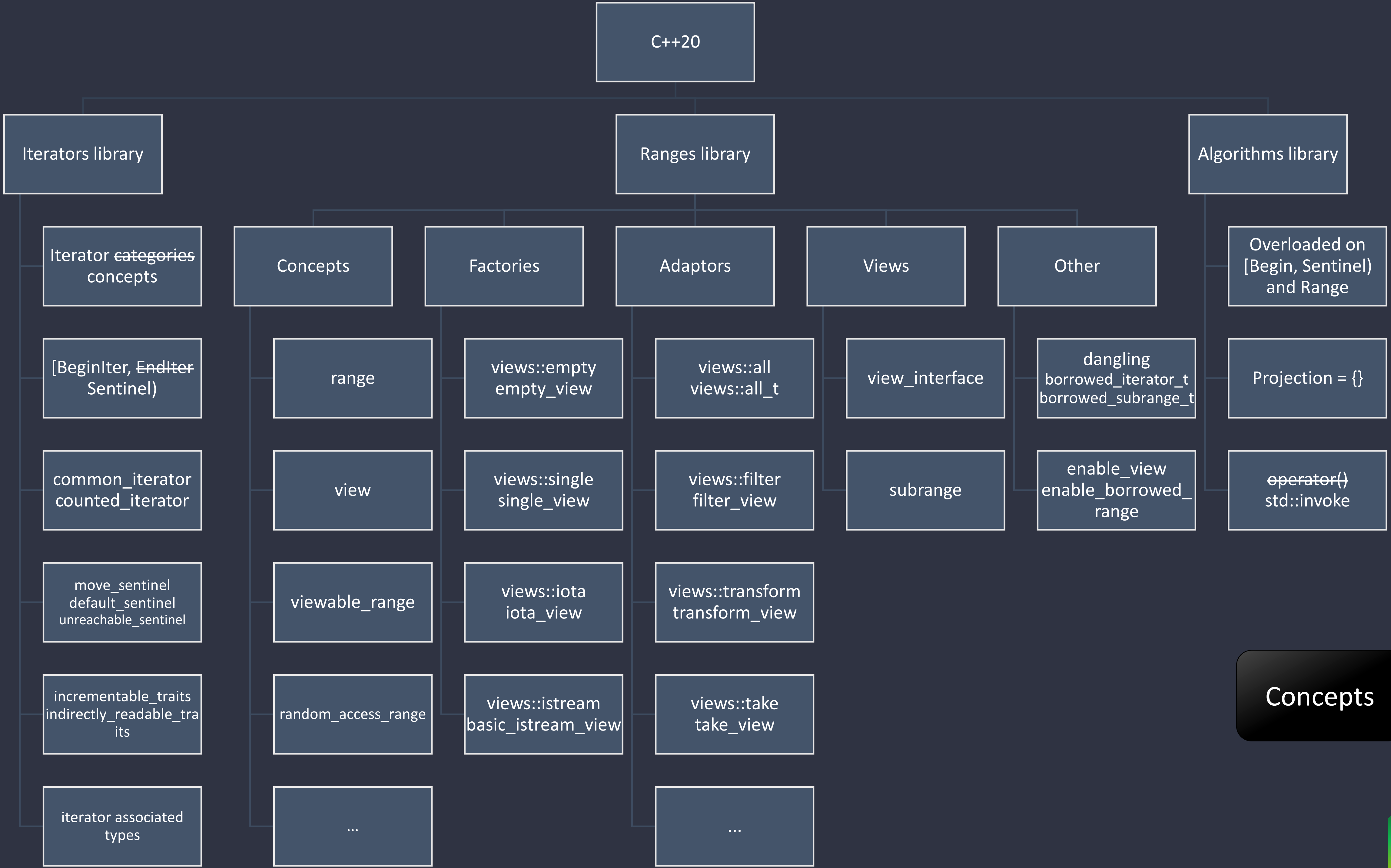
- range: something that can be iterated over
- range algo: algorithm that takes range
- view: lazy range thats cheap (to copy)
- range adaptor: make a range into a view

Jeff Garland

From STL to Ranges

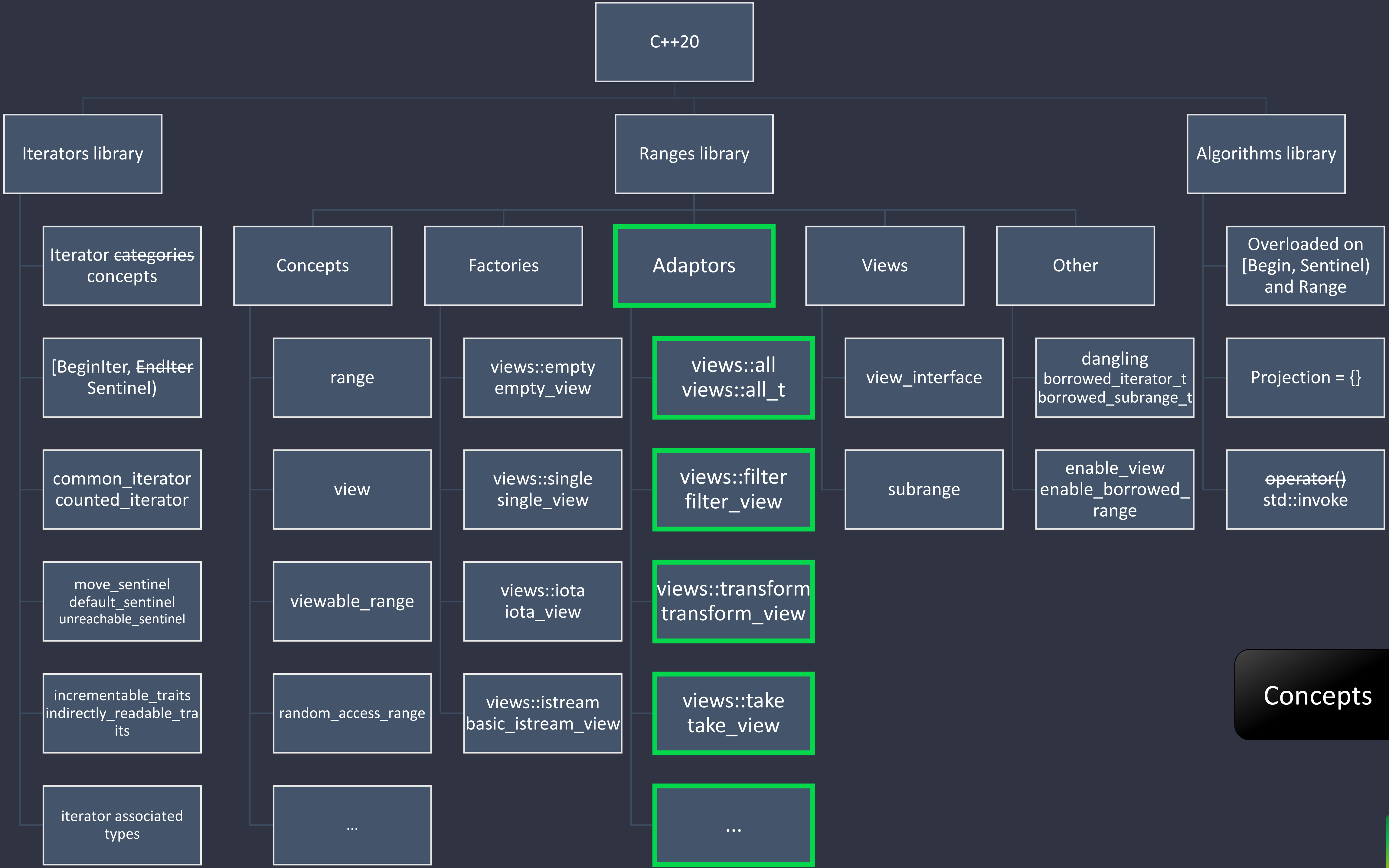


... but there was one thing that I did not get



- C++20
  - Iterators library
    - Iterator ~~categories~~ concepts
    - [BeginIter, ~~EndIter~~ Sentinel)
    - common_iterator counted_iterator
    - move_sentinel default_sentinel unreachable_sentinel
    - incrementable_traits indirectly_readable_traits
    - iterator associated types
  - Ranges library
    - Concepts
      - range
      - view
      - viewable_range
      - random_access_range
      - …
    - Factories
      - views::empty empty_view
      - views::single single_view
      - views::iota iota_view
      - views::istream basic_istream_view
    - Adaptors
      - views::all views::all_t
      - views::filter filter_view
      - views::transform transform_view
      - views::take take_view
      - …
    - Views
      - view_interface
      - subrange
    - Other
      - dangling borrowed_iterator_t borrowed_subrange_t
      - enable_view enable_borrowed_range
  - Algorithms library
    - Overloaded on [Begin, Sentinel) and Range
    - Projection = {}
    - ~~operator()~~ std::invoke

Concepts



- C++20
  - Iterators library
    - Iterator ~~categories~~ concepts
    - [BeginIter, ~~EndIter~~ Sentinel)
    - common_iterator
counted_iterator
    - move_sentinel
default_sentinel
unreachable_sentinel
    - incrementable_traits
indirectly_readable_traits
    - iterator associated types
  - Ranges library
    - Concepts
      - range
      - view
      - viewable_range
      - random_access_range
      - …
    - Factories
      - views::empty
empty_view
      - views::single
single_view
      - views::iota
iota_view
      - views::istream
basic_istream_view
    - Adaptors
      - views::all
views::all_t
      - views::filter
filter_view
      - views::transform
transform_view
      - views::take
take_view
      - …
    - Views
      - view_interface
      - subrange
    - Other
      - dangling
borrowed_iterator_t
borrowed_subrange_t
      - enable_view
enable_borrowed_range
  - Algorithms library
    - Overloaded on [Begin, Sentinel) and Range
    - Projection = {}
    - ~~operator()~~
std::invoke

Concepts

Standard example with one Range Adaptor Object

```
std::array a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
auto evenValues = a | std::views::filter(is_even);
for (int x : evenValues)
{
    std::cout << x << ' ';
}
```

Standard example with one Range Adaptor Object

```cpp
std::array a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
auto evenValues = a | std::views::filter(is_even);
for (int x : evenValues)
{
    std::cout << x << ' ';
}
```

```
Output: 4 2
```

Standard example with one Range Adaptor Object

```
std::array a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
auto evenValues = a | std::views::filter(is_even);
for (int x : evenValues)
{
    std::cout << x << ' ';
}
```

Output: 4 2

We naturally expect evenValues to hold a reference to the array a.

Standard example with two Range Adaptor Objects

```cpp
std::array a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
auto square = [](int i) { return i * i; };
auto squareOfEvenValues = a | std::views::filter(is_even)
                            | std::views::transform(square);

for (int x : squareOfEvenValues)
{
    std::cout << x << ' ';
}
```

Standard example with two Range Adaptor Objects

```
std::array a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
auto square = [](int i) { return i * i; };
auto squareOfEvenValues = a | std::views::filter(is_even)
                            | std::views::transform(square);

for (int x : squareOfEvenValues)
{
    std::cout << x << ' ';
}
```

Output: 16 4

My contradiction

```
a | std::views::filter(Pred);
```

Constructs a `filter_view`
holding a reference to a

```
a | std::views::filter(Pred) | std::views::transform(Func);
```

Constructs a `filter_view`
holding a reference to a

Constructs a `transform_view`
holding a copy of the `filter_view`

Going even further

```
auto is_even = [](int i) { return i % 2 == 0; };
auto evenValues = std::array{ 4, 5, 2, 7 } | std::views::filter(is_even);
for (int x : evenValues)
{
    std::cout << x << ' ';
}
```

Output: 4 2

This time `evenValues` has to store the array by value.

Concepts

- std::ranges::range
  - A range provides begin-iterator and end-sential, which denote a range [b, e)

```
static_assert(std::ranges::range<std::array<int, 5>>);
static_assert(std::ranges::range<std::string>);
static_assert(std::ranges::range<const char[14]>);
```

- std::ranges::view
  - Refinement of range
  - Needs to be movable. If it also is copyable, then copying needs to be cheap
  - More of a semantic requirement that it is cheap to pass it around by value

```
static_assert(std::ranges::view<std::string_view>);
static_assert(std::ranges::view<std::ranges::filter_view<...>>);
static_assert(std::ranges::view<std::filesystem::directory_iterator>);
```

Classes

- std::ranges::ref_view
  - Stores a range by reference
  - Like a std::reference_wrapper, but for ranges
- std::ranges::owning_view
  - Stores a range by value
  - Move only

```cpp
std::string text = "Amazing";
std::ranges::ref_view ref = text;
std::ranges::owning_view owner = std::string{"Feature"};
for (char c : ref  ) { std::cout << c; }   // Amazing
for (char c : owner) { std::cout << c; }   // Feature
```

My contradiction

```
a | std::views::filter(Pred);
```

Constructs a `filter_view`
holding a reference to a

```
a | std::views::filter(Pred) | std::views::transform(Func);
```

Constructs a `filter_view`
holding a reference to a

Constructs a `transform_view`
holding a copy of the `filter_view`

Code Transformation

<table>
<tr><td></td><td>a | std::views::filter(is_even)</td><td>a | std::views::filter(is_even)
  | std::views::transform(square)</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

Three ways of using Range Adaptor Objects

1. `VR | adaptor(Args...)`
2. `adaptor(Args...)(VR)`
3. `adaptor(VR, Args...)`

Code Transformation

	`a	std::views::filter(is_even)`	`a	std::views::filter(is_even)` `	std::views::transform(square)`

Rewriting Range Adaptor Objects	`std::views::filter(a, is_even)`	`std::views::transform(`
		`   std::views::filter(a, is_even),`
		`square)`

Expression-Equivalence between RAO and RAT

- The Range Adaptor Object `std::views::filter(VR, P)` is expression-equivalent to a constructor call of its Range Adaptor Type `std::ranges::filter_view(VR, P)`
- Same applies to `std::views::transform(VR, F)` and `std::ranges::transform_view(VR, F)`
- Not every RAO is expression-equivalent to a RAT (see: bonus slide)
- Expression-Equivalent
  - Same effects
  - Same noexcept-ness
  - Same constexpr-ness

Code Transformation

<table>
<tr><th></th><th>a | std::views::filter(is_even)</th><th>a | std::views::filter(is_even)
  | std::views::transform(square)</th></tr>
<tr><td>Rewriting Range Adaptor Objects</td><td>std::views::filter(a, is_even)</td><td>std::views::transform(
   std::views::filter(a, is_even),
square)</td></tr>
<tr><td>RAO expression-equivalent RAT</td><td>std::ranges::filter_view(a, is_even)</td><td>std::ranges::transform_view(
   std::ranges::filter_view(a, is_even),
square)</td></tr>
</table>

Class Template Argument Deduction

- CTAD is available since C++17
- Similar approach as for function template argument deduction
  - `std::min(1, 2)` calls `std::min<int>`
- Examples
  - `std::pair p(2, 4.5)` => deduces to `std::pair<int, double>`
  - `std::lock_guard lg(mtx)` => deduces to `std::lock_guard<std::mutex>` (or of whatever type `mtx` is)
  - `std::experimental::scope_exit atExit = [] { print("Thanks for listening!"); };` => deduces to `scope_exit<decltype(`*lambda expression*`)>;`
- We can provide custom Deduction Guides to tell the compiler how to deduce the final class from the provided arguments

Our first Deduction Guide

```
template<class T>
struct Text {
    Text(const T&);
    // ...
};


Text(const char*) -> Text<std::string_view>; // This is a so-called user-defined deduction guide


Text sample("Hello"); // deduces to Text<std::string_view>
```

Our second Deduction Guide

```
template<class T>
struct Holder {
    Holder(T);
    // ...
};


template<class X>
using Box = std::conditional_t<
    std::is_lvalue_reference_v<X>, // Condition
    std::reference_wrapper<std::remove_reference_t<X>>, // True-Type
    X // False-Type
>;


template<class U>
Holder(U&&) -> Holder<Box<U>>;


Holder a(1.2); // deduces to ???
std::string s;
Holder b(s); // deduces to ???
```

Our second Deduction Guide

```
template<class T>
struct Holder {
    Holder(T);
    // ...
};


template<class X>
using Box = std::conditional_t<
    std::is_lvalue_reference_v<X>, // Condition
    std::reference_wrapper<std::remove_reference_t<X>>, // True-Type
    X // False-Type
>;


template<class U>
Holder(U&&) -> Holder<Box<U>>;


Holder a(1.2); // deduces to Holder<double>. Instantiates the deduction guide with U = double
std::string s;
Holder b(s); // deduces to Holder<std::reference_wrapper<std::string>>
                                             Instantiates the deduction guide with U = std::string&
```

Our last Deduction Guide

```
template<class T>
struct MyRangeAdaptor {
    MyRangeAdaptor(T);
    // ...
};


template<class X>
using MyAll_t = std::conditional_t<
    std::ranges::view<std::remove_cvref_t<X>>, // Is X a view?
    std::remove_cvref_t<X>, // Then MyAll_t is X
    std::conditional_t<std::is_lvalue_reference_v<X>, // Else: Is X an lvalue reference?
        std::ranges::ref_view<std::remove_reference_t<X>>, // Then MyAll_t is ref_view<X>
        std::ranges::owning_view<std::remove_reference_t<X>> // Otherwise MyAll_t is owning_view<X>
    >
>;

template<class R>
MyRangeAdaptor(R&&) -> MyRangeAdaptor<MyAll_t<R>>;


MyRangeAdaptor a = std::string_view{"Hey, Ranges"}; // deduces to ???
std::array arr{ 4, 5, 2, 7 };
MyRangeAdaptor b(arr); // deduces to ???
MyRangeAdaptor c(std::string{"Hey, emBO++"}); // deduces to ???
```

Our last Deduction Guide

```
template<class T>
struct MyRangeAdaptor {
    MyRangeAdaptor(T);
    // ...
};


template<class X>
using MyAll_t = std::conditional_t<
    std::ranges::view<std::remove_cvref_t<X>>, // Is X a view?
    std::remove_cvref_t<X>, // Then MyAll_t is X
    std::conditional_t<std::is_lvalue_reference_v<X>, // Else: Is X an lvalue reference?
        std::ranges::ref_view<std::remove_reference_t<X>>, // Then MyAll_t is ref_view<X>
        std::ranges::owning_view<std::remove_reference_t<X>> // Otherwise MyAll_t is owning_view<X>
    >
>;

template<class R>
MyRangeAdaptor(R&&) -> MyRangeAdaptor<MyAll_t<R>>;


MyRangeAdaptor a = std::string_view{"Hey, Ranges"}; // deduces to MyRangeAdaptor<string_view>
std::array arr{ 4, 5, 2, 7 };
MyRangeAdaptor b(arr); // deduces to MyRangeAdaptor<std::ranges::ref_view<array<int, 4>>>
MyRangeAdaptor c(std::string{"Hey, emBO++"}); // deduces to MyRangeAdaptor<std::ranges::owning_view<string>>
```

Deduction Guides for Range Adaptor Types

```
template<class R, class Pred>
filter_view(R&&, Pred) -> filter_view<std::views::all_t<R>, Pred>;

template<class R, class F>
transform_view(R&&, F) -> transform_view<std::views::all_t<R>, F>;
```

Putting it all together

```
template<class R, class Pred> filter_view(R&&, Pred) -> filter_view<std::views::all_t<R>, Pred>;
template<class R, class F>    transform_view(R&&, F) -> transform_view<std::views::all_t<R>, F>;
```

Putting it all together

```
template<class R, class Pred> filter_view(R&&, Pred) -> filter_view<std::views::all_t<R>, Pred>;
template<class R, class F>    transform_view(R&&, F) -> transform_view<std::views::all_t<R>, F>;


using A = std::array<int, 4>;
A a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
using Pred = decltyp(is_even);
auto square = [](int i) { return i * i; };
using F = decltyp(square);
```

Putting it all together

```
template<class R, class Pred> filter_view(R&&, Pred) -> filter_view<std::views::all_t<R>, Pred>;
template<class R, class F>    transform_view(R&&, F) -> transform_view<std::views::all_t<R>, F>;


using A = std::array<int, 4>;
A a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
using Pred = decltyp(is_even);
auto square = [](int i) { return i * i; };
using F = decltyp(square);

auto evenValues = a | std::views::filter(is_even);



using FilterViewType = decltype(evenValues);
auto squareOfEvenValues = a | std::views::filter(is_even) | std::views::transform(square);
```

Putting it all together

```
template<class R, class Pred> filter_view(R&&, Pred) -> filter_view<std::views::all_t<R>, Pred>;
template<class R, class F>    transform_view(R&&, F) -> transform_view<std::views::all_t<R>, F>;


using A = std::array<int, 4>;
A a{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
using Pred = decltyp(is_even);
auto square = [](int i) { return i * i; };
using F = decltyp(square);

auto evenValues = a | std::views::filter(is_even);
        //rewrite adaptor:         std::views::filter(a, is_even)
        //expression-equivalent:   filter_view(a, is_even)
        //CTAD & deduction guide:  filter_view<std::views::all_t<A&>, Pred>
        //finally:                 filter_view<std::ranges::ref_view<A>, Pred>
using FilterViewType = decltype(evenValues);
auto squareOfEvenValues = a | std::views::filter(is_even) | std::views::transform(square);
        //rewrite adaptor:         std::views::transform(views::filter(a, is_even), square)
        //expression-equivalent:   transform_view(filter_view(a, is_even), square)
        //CTAD & deduction guide:  transform_view<std::views::all_t<FilterViewType>, F>
        //finally:                 transform_view<FilterViewType, F>
```

Summary

- Most Range Adaptor Objects are expression - equivalent to their according Range Adaptor Type (`std::views::transform -> std::ranges::transform_view`)
- Range Adaptors have a Deduction Guide that uses forwarding references and `std::views::all_t<R>`
- `std::views::all(VR)` turns a `viewable_range` into a `view`
  - If `VR` already is a `view` (e.g., `std::string_view`) => return it
  - Else if `VR` is an lvalue (e.g., `std::array<int, 4>&`) => return `std::ranges::ref_view(VR)`
  - Else (e.g., `CreateArray()`) => `std::ranges::owning_view(VR)`
- This explains the different behavior between:
  - `a | std::views::filter(is_even)`
  - `a | std::views::filter(is_even) | std::views::transform(square)`

Two Bonus Slides on

std::views::XXX
VS
std::ranges::XXX_view

Relation between views::XXX and ranges::XXX_view

Expression-Equivalent	
views::filter	ranges::filter_view
views::transform	ranges::transform_view
views::take_while	ranges::take_while_view
views::drop_while	ranges::drop_while_view
views::split	ranges::split_view
views::lazy_split	ranges::lazy_split_view

views::join	ranges::join_view
join on a join_view will return a join_view<join_view>	Calls copy constructor on join_view

views::elements<N>	ranges::elements_view<R, N>
views::keys	ranges::key_view<R>
views::values	ranges::values_view<R>

views::all, views::all_t

- A view over all elements of a viewable_range
- views::all_t = decltype(views::all(...))

ranges::ref_view

- Like reference_wrapper but for a range

ranges::owning_view

- Move-only owner of a range

views::take, ranges::take_view

- e.g. views::take(„ABCDE“sv, 3) is „ABC“sv

views::drop, ranges::drop_view

- e.g. views::drop(„ABCDE“sv, 3) is „DE“sv

views::counted

- view over [It, It + N)

views::common, ranges::common_view

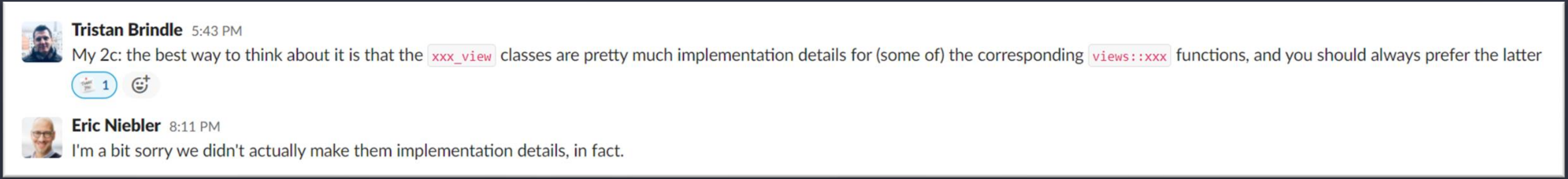
- views::all(VR) if VR models common_range
- common_view(VR) otherwise

views::reverse, ranges::reverse_view

- Special treatment of reverse_view and reverse_iterator

Asking the Experts

- Taken from cpplang.slack.com (channel: #ranges, date: November 3rd, 2020)



Tristan Brindle 5:43 PM
My 2c: the best way to think about it is that the `xxx_view` classes are pretty much implementation details for (some of) the corresponding `views::xxx` functions, and you should always prefer the latter

Eric Niebler 8:11 PM
I'm a bit sorry we didn't actually make them implementation details, in fact.

```
std::views::take(
   Questions, n)
```