Deduction Guides for Range Adaptors

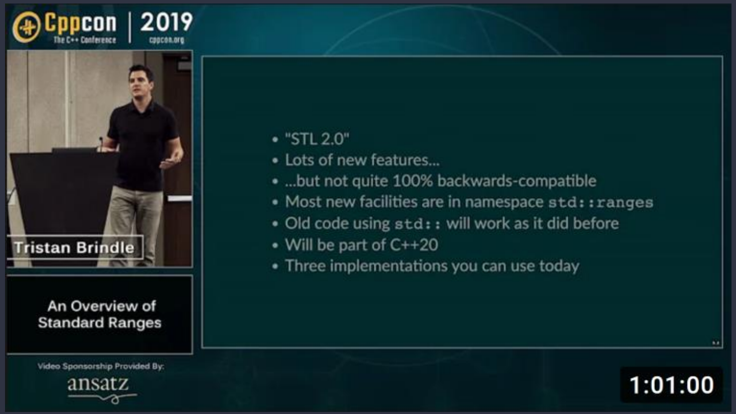
Kilian Henneberger

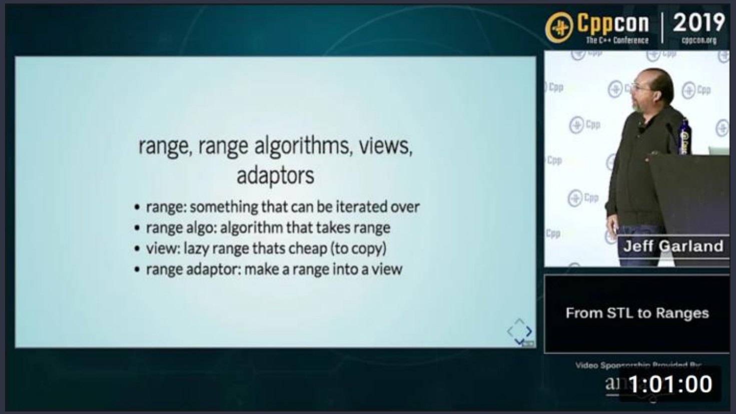
January 12, 2022

Ranges Library

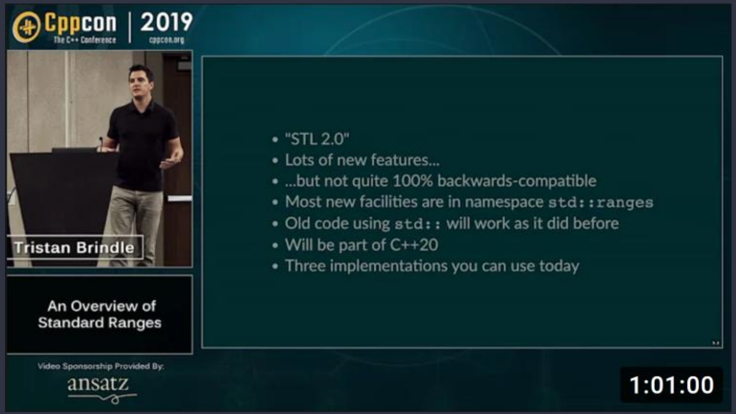
- C++20 added <ranges>
- Documented on cppreference.com
- Earliest available since
  - GCC libstdc++ 10.1 (May 7, 2020)
  - MSVC STL in VS2019 16.6 (July 15, 2020)
  - Clang libc++ 13.0.0 (Oct 4, 2021), In Progress
- C++23 will add further changes and features
- The C++ Standard, cppreference and the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

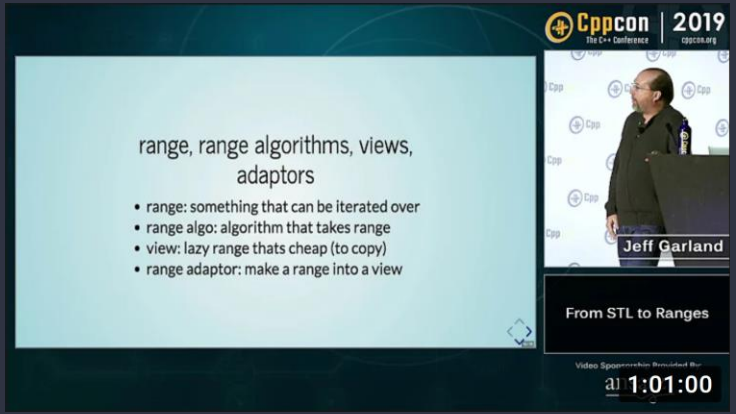
A lot of good talks...

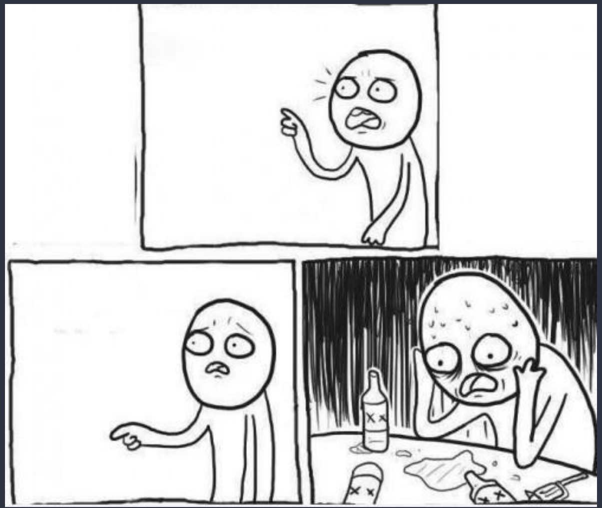




A lot of good talks...







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

- C++20
  - Iterators library
    - Iterator ~~categories~~ concepts
    - [Begin, ~~End~~ 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
    - [Begin, ~~End~~ 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

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

Standard example with one Range Adaptor Object

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

Output: 4 2

Standard example with one Range Adaptor Object

```
std::vector<int> v{ 4, 5, 2, 7 };
auto is_even = [](int i) { return i % 2 == 0; };
auto evenValues = v | 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 vector v.

Standard example with two Range Adaptor Objects

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

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

Standard example with two Range Adaptor Objects

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

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

Output: 16 4

My contradiction

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

Constructs a `filter_view`
holding a reference to `v`

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

Constructs a `filter_view`
holding a reference to `v`

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

Code Transformation

	`v | std::views::filter(is_even)`	`v | std::views::filter(is_even)` `| std::views::transform(square)`

Three ways of using Range Adaptor Objects

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

Code Transformation

	`v | std::views::filter(is_even)`	`v | std::views::filter(is_even)` `  | std::views::transform(square)`

Rewriting Range Adaptor Objects	`std::views::filter(v, is_even)`	`std::views::transform(`
		`  std::views::filter(v, 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 its RAT (see: bonus slide)
- Expression-Equivalent
  - Same effects
  - Same noexcept-ness
  - Same constexpr-ness

Code Transformation

	`v | std::views::filter(is_even)`	`v | std::views::filter(is_even)` `| std::views::transform(square)`

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

RAO expression-equivalent RAT	`std::ranges::filter_view(v, is_even)`	`std::ranges::transform_view(`
		`std::ranges::filter_view(v, is_even),`
		`square)`

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)
- 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 MyReverseView {
    MyReverseView(T);
    // ...
};


template<class X>
using MyAll_t = std::conditional_t<
    std::ranges::view<std::remove_cvref_t<X>>, // Condition
    std::remove_cvref_t<X>, // True-Type
    std::ranges::ref_view<std::remove_reference_t<X>> // False-Type
>;


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


MyReverseView a = std::string_view{"Hello Ranges"}; // deduces to ???
std::vector<int> v{ 4, 5, 2, 7 };
MyReverseView b(v); // deduces to ???
```

Our last Deduction Guide

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


template<class X>
using MyAll_t = std::conditional_t<
    std::ranges::view<std::remove_cvref_t<X>>, // Condition
    std::remove_cvref_t<X>, // True-Type
    std::ranges::ref_view<std::remove_reference_t<X>> // False-Type
>;


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


MyReverseView a = std::string_view{"Hello Ranges"}; // deduces to MyReverseView<std::string_view>
std::vector<int> v{ 4, 5, 2, 7 };
MyReverseView b(v); // deduces to MyReverseView<std::ranges::ref_view<std::vector<int>>>
```

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 V = std::vector<int>;
V v{ 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 V = std::vector<int>;
V v{ 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 = v | std::views::filter(is_even);




using FilterViewType = decltype(evenValues);
auto squareOfEvenValues = v | 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 V = std::vector<int>;
V v{ 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 = v | std::views::filter(is_even);
        //rewrite adaptor:           std::views::filter(v, is_even)
        //expression-equivalent:     filter_view(v, is_even)
        //CTAD & deduction guide:    filter_view<std::views::all_t<V&>, Pred>
        //finally:                   filter_view<std::ranges::ref_view<V>, Pred>
using FilterViewType = decltype(evenValues);
auto squareOfEvenValues = v | std::views::filter(is_even) | std::views::transform(square);
        //rewrite adaptor:           std::views::transform(views::filter(v, is_even), square)
        //expression-equivalent:     transform_view(filter_view(v, 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::vector<int>&`) => return `std::ranges::ref_view(VR)`
  - Else (e.g. `CreateVector()`) => `std::ranges::owning_view(VR)`
- This explains the different behavior between:
  - `v | std::views::filter(is_even)`
  - `v | 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)

tcbrindle 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
1

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

arthur-odwyer 10:46 PM
I was gonna say, is there any rationale for why the names of the view classes shouldn't be implementation-defined?
Vice versa, is it reasonable to teach that iota_view(m,n) is bad style in the same sense as make_pair<int,int>(1,2) is bad style?

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