RANGE ALGORITHMS, VIEWS AND ACTIONS

A COMPREHENSIVE GUIDE

DVIR YITZCHAKI

CPPCON 2019

Getty Images

DVIR YITZCHAKI

- Sr. software engineer at verizon media
- Casual speaker at Core C++
- dvirtz at GitHub slack Gmail
- @dvirtzwastaken on twitter
- Likes going to music concerts

THANK YOU

- Christopher Di Bella
- Adi Shavit

JACK GUEZ/AFP/GETTY IMAGES

I CAN'T GET NO

MOTIVATION

WHAT IS A RANGE

- A sequence of elements between two locations i, k.
- Often denoted by [i, k)

WHAT IS A RANGE

- A pair of iterators?

```
std::copy(v.begin(), v.end(), buf);
```

- An iterator and a count of elements?

```
std::copy_n(v.begin(), 20, buf);
```

- An iterator and a predicate?

```
std::copy(std::istream_iterator<int>{std::cin},
          std::istream_iterator<int>{},
          buf);
```

- Why not all?

MOTIVATION

- Simpler syntax.
- Rather than this:

```
std::vector<int> v { /*...*/ };
std::sort(v.begin(), v.end());
```

write this:

```
std::ranges::sort(v);
```

- New syntax enables composition:

```
accumulate(views::iota(1)
  | views::transform([](int x) { return x * x; })
  | views::take(10), 0);
```

DREAM OF

IMPLEMENTATION

CMCSTL2

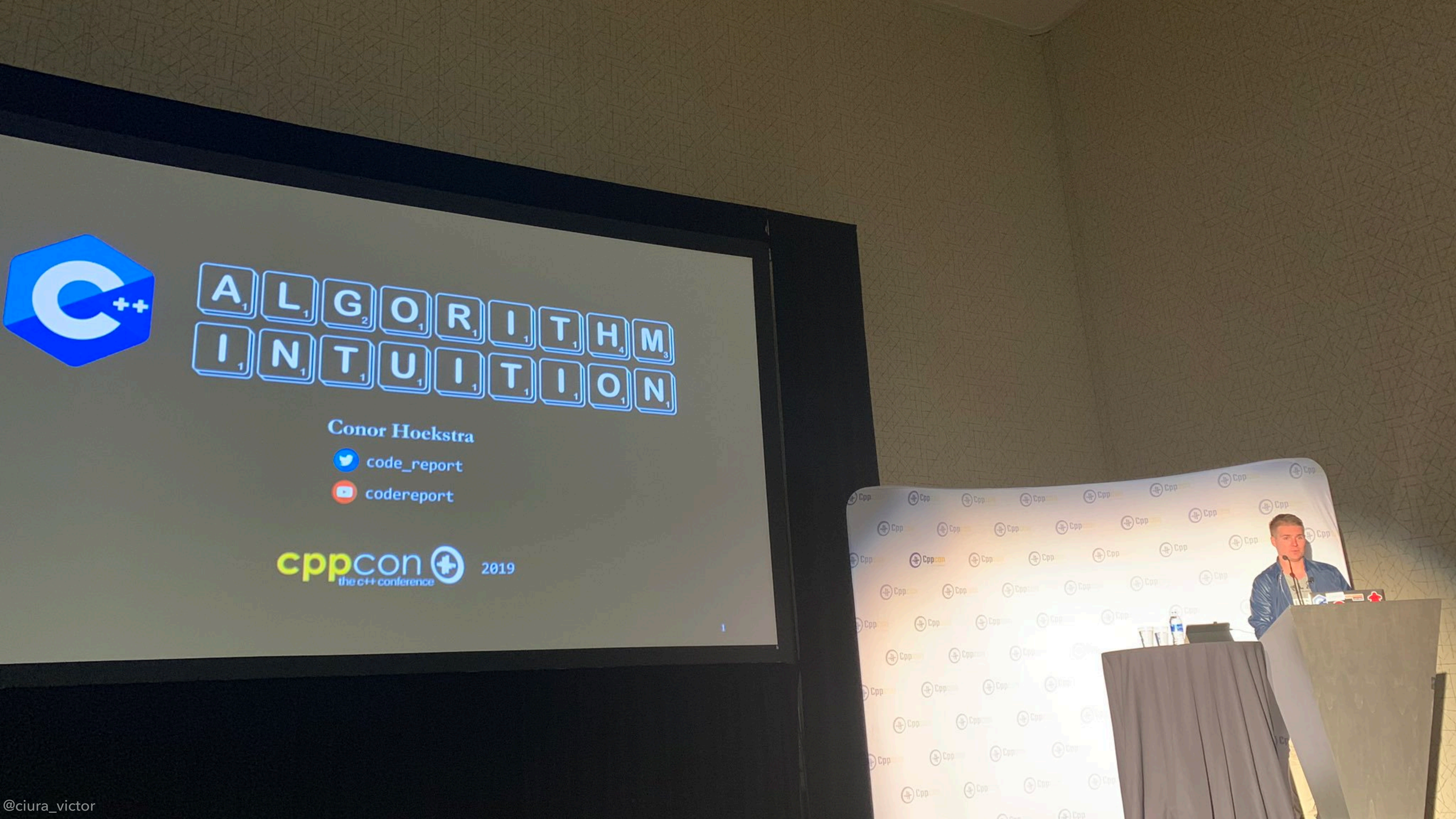
- By Casey Carter
- https://github.com/CaseyCarter/cmcstl2
- Reference implementation of the standard
- Requires GCC 7+ with -std=c++1z -fconcepts

NANO RANGE

- By Tristan Brindle
- https://github.com/tcbrindle/NanoRange
- C++17 implementation of the C++20 Ranges
- Minimum compiler versions:
  - GCC 7
  - Clang 4.0
  - MSVC 2017 version 15.9 with /permissive-

RANGE-V3

- By Eric Niebler
- https://github.com/ericniebler/range-v3
- Everything that is C++20 and much more
- C++14 with optional features for C++17/20
- Emulated concepts if not available in compiler
- Minimum compiler versions:
  - clang 3.6.2 (or later)
  - GCC 5.0.2 (or later)
  - Visual Studio 2019 with /std:c++17 /permissive- /experimental:preprocessor



AP/Bernat Armangue

BABY YOU CAN DRIVE MY

CONCEPTS

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr ? for_each(InputIt first, InputIt last, UnaryFunction f);
```

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

ITERATORS

- **`input_or_output_iterator`** - can be dereferenced (`*it`) and incremented (`++it`)
- **`input_iterator`** - referenced values can be read (`auto v = *it`)
- **`output_iterator`** - referenced values can be written to (`*it = v`)
- **`forward_iterator`** - `input_iterator` + comparable and multi-pass
- **`bidirectional_iterator`** - `forward_iterator` + decrementable (`--it`)
- **`random_access_iterator`** - `bidirectional_iterator` + random access (`it += n`)
- **`contiguous_iterator`** - `random_access_iterator` + contiguous in memory

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

SENTINELS – THE NEW END ITERATOR

- sentinel_for - a semi-regular S is a sentinel_for an iterator I if:
  - Let s and i be values of type S and I, respectively, such that [i, s) denotes a range
  - i == s is well defined
  - If i != s then i is dereferenceable and
    [++i, s) denotes a range
- Such an s is called a sentinel

SENTINELS – UNIFING CONSTRUCT

- Examples:
  - **Pair of iterators**
  - **Iterator and predicate**
  - **Iterator and count**

RANGE

- **`range`** - a type we can feed to
  - `ranges::begin` - to get an iterator
  - `ranges::end` - to get a sentinel
- A **range** `[i, s)` refers to the elements

  `*i, *(i++), *((i++)++), …, j`

  such that `j == s`.

RANGE CONCEPTS

- **`input_range`** - e.g. a range over a `std::istream_iterator`
- **`output_range`** - e.g. a range over a `std::back_insert_iterator`
- **`forward_range`** - e.g. `std::forward_list`
- **`bidirectional_range`** - e.g. `std::list`
- **`random_access_range`** - e.g. `std::deque`
- **`contiguous_range`** - e.g. `std::vector`
- **`common_range`** - sentinel is same type as iterator
  - e.g. standard containers

OTHER HELPFUL CONCEPTS & CALLABLES

- **`sized_sentinel`** - get distance by `s - i`.
- **`sized_range`** - get distance by **`ranges::size`**
  - Not necessarily implies `sized_sentinel` (`std::list`)
- **`ranges::distance`** - get size of range (can have linear complexity)
- **`ranges::empty`** - checks if a range has no elements
- **`ranges::data`** - gives a pointer to the data of a contiguous range

SATURDAY NIGHT'S ALRIGHT FOR ALGORITHMS

ranges::for_each

```
namespace ranges {

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
           indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = {});

  template<input_range R, class Proj = identity,
           indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
           indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = {});

  template<input_range R, class Proj = identity,
           indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
           indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = {});

  template<input_range R, class Proj = identity,
           indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
           indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = {});

  template<input_range R, class Proj = identity,
           indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::for_each

```
namespace ranges {

  template<class I, class F>
  struct for_each_result {
    [[no_unique_address]] I in;
    [[no_unique_address]] F fun;

    template<class I2, class F2>
      requires convertible_to<const I&, I2> && convertible_to<const F&, F2>
      operator for_each_result<I2, F2>() const & {
        return {in, fun};
      }

    template<class I2, class F2>
      requires convertible_to<I, I2> && convertible_to<F, F2>
      operator for_each_result<I2, F2>() && {
        return {std::move(in), std::move(fun)};
      }
  };

}
```

ranges::for_each

```
namespace ranges {

  template<class I, class F>
  struct for_each_result {
    [[no_unique_address]] I in;
    [[no_unique_address]] F fun;

    template<class I2, class F2>
      requires convertible_to<const I&, I2> && convertible_to<const F&, F2>
      operator for_each_result<I2, F2>() const & {
        return {in, fun};
      }

    template<class I2, class F2>
      requires convertible_to<I, I2> && convertible_to<F, F2>
      operator for_each_result<I2, F2>() && {
        return {std::move(in), std::move(fun)};
      }
  };

}
```

EXAMPLE

```
1   std::vector<int> v1{0, 2, 4, 6};
2   int sum  = 0;
3   auto &&[i, f] = for_each(v1, [&](int i) { sum += i; });
4   assert(sum == 12);
5   assert(i == v1.end());
6   f(1);
7   assert(sum == 13);
```

NOTE

All code examples and more can be found at:

https://github.com/dvirtz/ranges_code_samples

Tested using `gcc-8 -std=c++2a -fconcepts` with:

- range-v3
- cmcstl2

They assume `using namespace ranges;`

ranges::for_each

```
namespace ranges {

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
           indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = {});

  template<input_range R, class Proj = identity,
           indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, Fun f, Proj proj = {});
}
```

ranges::dangling

```
std::vector<int> f();

auto result1 = find(f(), 42);
static_assert(std::is_same_v<decltype(result1), dangling>);
// *result1 does not compile

auto vec     = f();
auto result2 = find(vec, 42);
static_assert(std::is_same_v<decltype(result2),
  std::vector<int>::iterator>);
```

It's not just for_each

JONATHAN BOCCARA

105 STL Algorithms in Less Than an Hour


Value Queries
transform_inclusive_scan
Valley of Partial Sums
transform_exclusive_scan
inclusive_scan
exclusive_scan
partial_sum
Province of Property Queries
2-Ranges Properties
none_of
equal
mismatch
lexicographical compare
is_permutation
set_symmetric_difference
Glorious County of Algos on Sets
set_intersection
set_union
includes
merge
for_each
transform
Lonely Islands
uninitialized_value_construct
uninitialized_default_construct
uninitialized_copy
uninitialized_fill
uninitialized_move
destroy
copy_backward
Lands of Queries
find
adjacent_find
transform_reduce
Value Searchers
Lagoon of Folds
reduce
equal_range
upper_bound
inner_product
accumulate
binary_search
lower_bound
sample
adjacent_difference
Peninsula of Raw Memory
Territory of Movers
copy
move
swap_ranges
move_backward
Province of Research
max_element
min_element
Relative Value Searchers
minmax_element
find_first_of
Bay of Range Searches
find_end
search
remove
unique
Island of Structure Changers
iota
fill
generate
replace
Land of Value Modifiers
Secret Runes
Lands of Permutations
partition_point
partition
reverse
rotate
sort
partial_sort
Shore of Sorting
sort_heap
nth_element
inplace_merge
make_heap
pop_heap
next_permutation
stable_*
The World of C++ STL Algorithms
Fluent C++

RANGE ALGORITHMS NOT IN C++20

- <numerics>
- parallel algorithms
- algorithms added after Ranges TS:

```
for_each_n      shift_left      search(range, searcher)
clamp           shift_right
sample          lexicographical_compare_three_way
```

GOD ONLY KNOWS WHAT I'D DO WITHOUT

VIEWS

Brian Wilson's PR

CONCEPTS

- **`view`** - a `range` type that has **constant time** copy, move, and assignment operators.
- Examples:
  - A Range type that wraps a pair of iterators.
  - A Range type that holds its elements by shared_ptr and shares ownership with all its copies.
  - A Range type that generates its elements on demand
  - Note: most containers are NOT views.

CONCEPTS

- **`viewable_range`** - a **`range`** type that can be converted to a view safely.
  - Examples:
    - views
    - **lvalue** containers

EXAMPLE

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
    static constexpr T* begin() noexcept { return nullptr; }
    static constexpr T* end() noexcept { return nullptr; }
    static constexpr T* data() noexcept { return nullptr; }
    static constexpr size_t size() noexcept { return 0; }
    static constexpr bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

EXAMPLE

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
    static constexpr T* begin() noexcept { return nullptr; }
    static constexpr T* end() noexcept { return nullptr; }
    static constexpr T* data() noexcept { return nullptr; }
    static constexpr size_t size() noexcept { return 0; }
    static constexpr bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

EXAMPLE

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
    static constexpr T* begin() noexcept { return nullptr; }
    static constexpr T* end() noexcept { return nullptr; }
    static constexpr T* data() noexcept { return nullptr; }
    static constexpr size_t size() noexcept { return 0; }
    static constexpr bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

EXAMPLE

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
    static constexpr T* begin() noexcept { return nullptr; }
    static constexpr T* end() noexcept { return nullptr; }
    static constexpr T* data() noexcept { return nullptr; }
    static constexpr size_t size() noexcept { return 0; }
    static constexpr bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

EXAMPLE

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
  class empty_view : public view_interface<empty_view<T>> {
  public:
    static constexpr T* begin() noexcept { return nullptr; }
    static constexpr T* end() noexcept { return nullptr; }
    static constexpr T* data() noexcept { return nullptr; }
    static constexpr size_t size() noexcept { return 0; }
    static constexpr bool empty() noexcept { return true; }

    friend constexpr T* begin(empty_view) noexcept { return nullptr; }
    friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```

FACTORIES & ADAPTORS

- Utility objects for creating views
- Called adaptors (if transform an existing range) or factories (otherwise)
- For example:

```
namespace std::ranges {
  namespace views {
    template<class T>
      inline constexpr empty_view<T> empty{};
  }
}
```

LAZINESS

- Views are lazily evaluated
- They generate their elements only on demand, when iterated.
- For this reason, there can be views with infinitely many elements.

NOTE

- The code examples to follow use the following function:

```
template<ranges::range V, typename T>
void check_equal(V &&v, std::initializer_list<T> il) {
    assert(ranges::equal(std::forward<V>(v), il));
}
```

- The views that are available in C++20 are denoted by  .

GENERATORS

`views::empty`

```
assert(empty(views::empty<int>));
```

GENERATORS

`views::single`

42

```
check_equal(views::single(42), {42});
```

GENERATORS

subrange

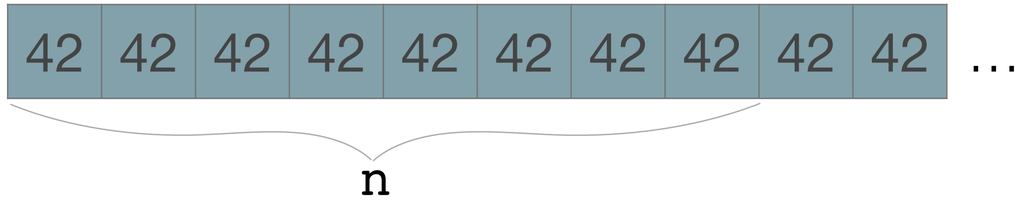
iterator + sentinel → view

```
const int rng[] = {1, 2, 3, 4};
check_equal(subrange{begin(rng) + 1, end(rng)},
  {2, 3, 4}); }
```

GENERATORS

`views::repeat`

`views::repeat_n`

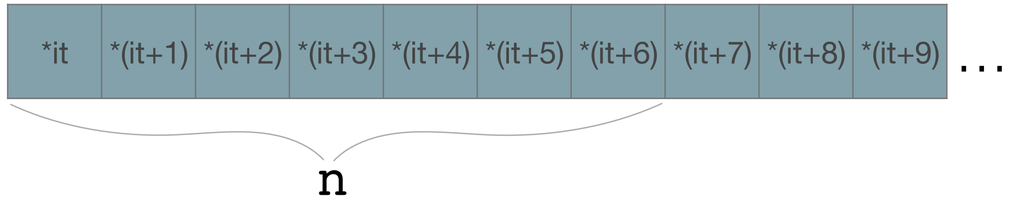



```
check_equal(views::repeat_n(42, 6), {42, 42, 42, 42, 42, 42});
```

GENERATORS

`views::unbounded`

`views::counted`

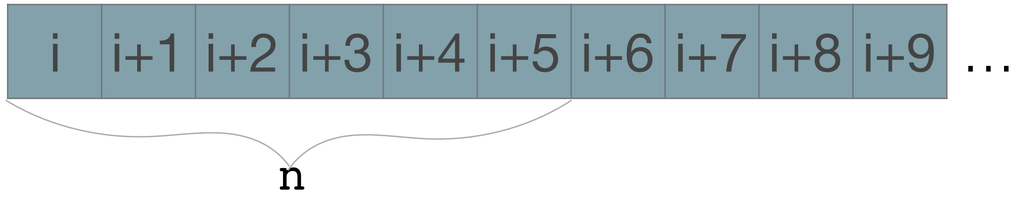



```
int rng[]  = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::counted(begin(rng) + 2, 5), {3, 4, 5, 6, 7});
```

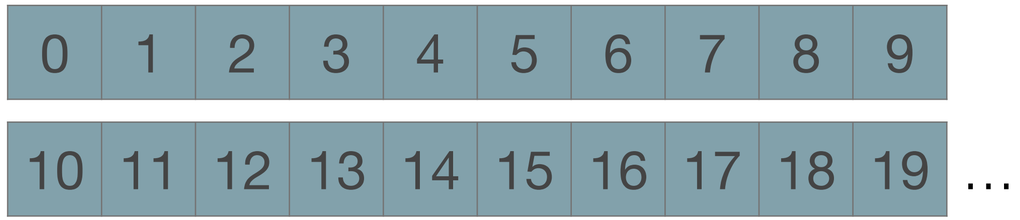
GENERATORS

```
views::iota(i)
```

```
views::iota(i, i + n)
```

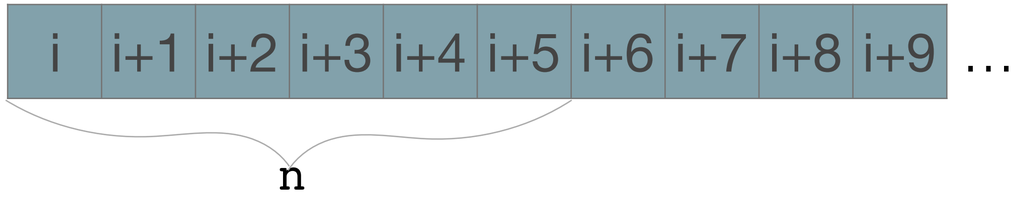



```
check_equal(views::iota(42, 45), {42, 43, 44});

check_equal(views::closed_iota(42, 45), {42, 43, 44, 45});

views::ints == views::iota<integral>

views::closed_indices == views::closed_iota<integral>

views::indices(10)

views::ints(10)
```



GENERATORS

```
views::iota(i, unreachable)
```



```
views::iota(i, i + n)
```

```
check_equal(views::iota(42, 45), {42, 43, 44});

check_equal(views::closed_iota(42, 45), {42, 43, 44, 45});

views::ints == views::iota<integral>

views::closed_indices == views::closed_iota<integral>

views::indices(10)
```


```
views::ints(10, unreachable)
```

GENERATORS

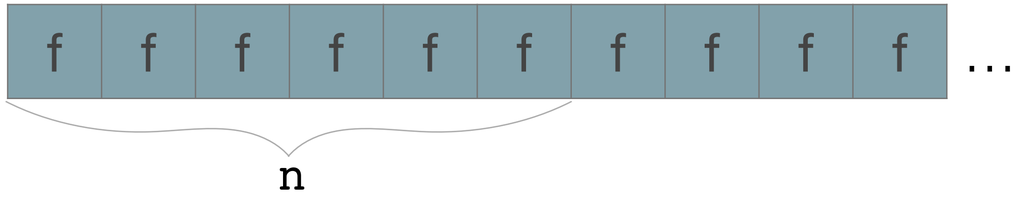
`views::linear_distribute`


```
check_equal(views::linear_distribute(0.5, 1.4, 10),
  {0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4});
```

GENERATORS

`views::generate`

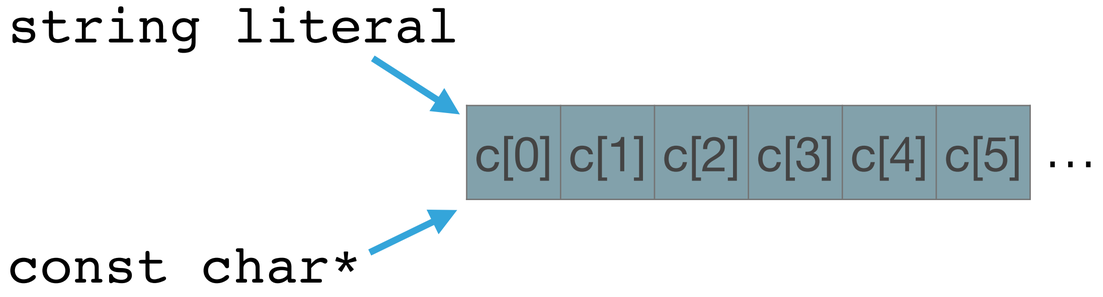
`views::generate_n`




```
check_equal(views::generate_n([i = 0]() mutable { return i++; }, 5),
  {0, 1, 2, 3, 4});
```

STRINGS

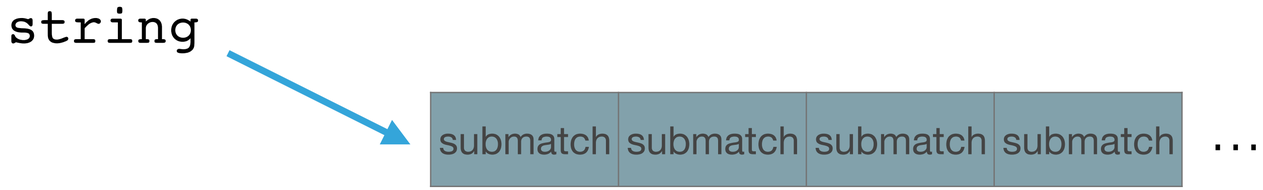
`views::c_str`




```
check_equal(views::c_str("cppcon"), {'c', 'p', 'p', 'c', 'o', 'n'});
```

STRINGS

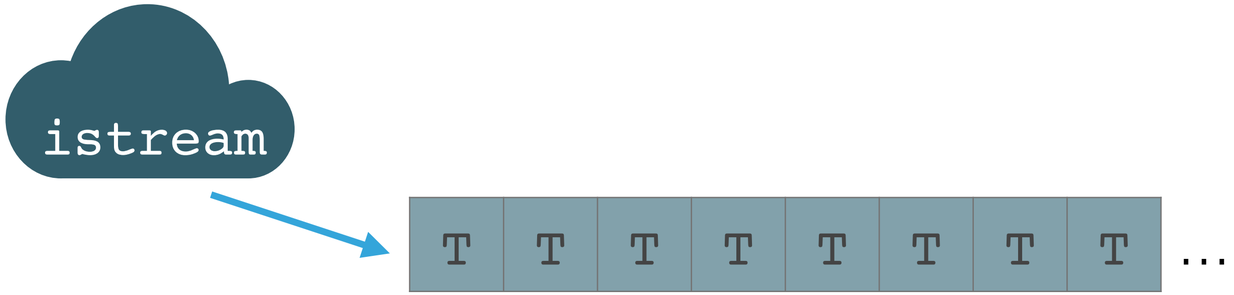
`views::tokenize`




```
auto str = "abc\ndef\tghi klm”s;
check_equal(views::tokenize(str, std::regex{"([a-z]+)"}, 1),
  {"abc"s, "def"s, "ghi"s, "klm"s});
```

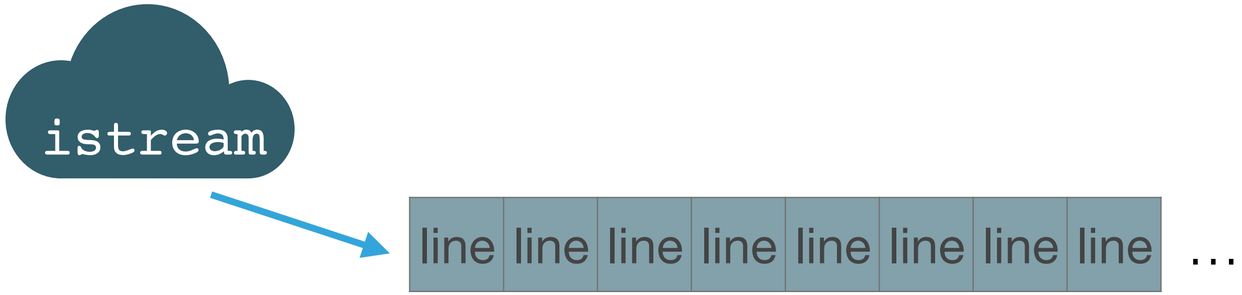
STREAMS

`istream<T>`




```
std::istringstream sst{"1 2 3 4 5 6”};
check_equal(istream<int>(sst), {1, 2, 3, 4, 5, 6});
```

STREAMS




```
getlines
```

```
std::istringstream sst{R"(One Proposal to
ranges::merge them all,
One Proposal to
ranges::find them)"};
check_equal(getlines(sst), {"One Proposal to",
"ranges::merge them all,",
"One Proposal to",
"ranges::find them"});
```

FILTERS

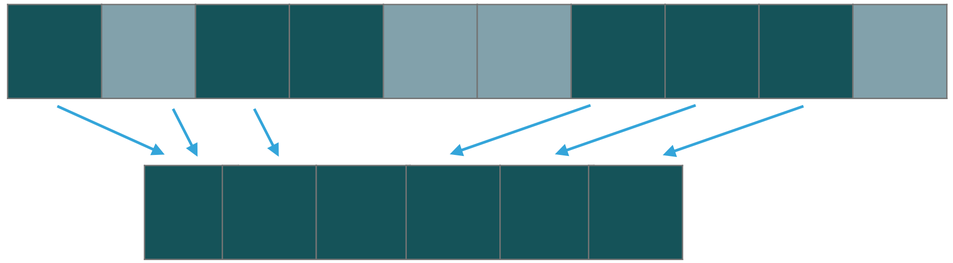
FILTERS

`views::filter`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::filter(rng, [](int i) { return i % 2 == 0; }),
  {0, 2, 4, 6, 8, 10});
```

FILTERS

`views::filter`



```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::filter(rng, [](int i) { return i % 2 == 0; }),
  {0, 2, 4, 6, 8, 10});
```

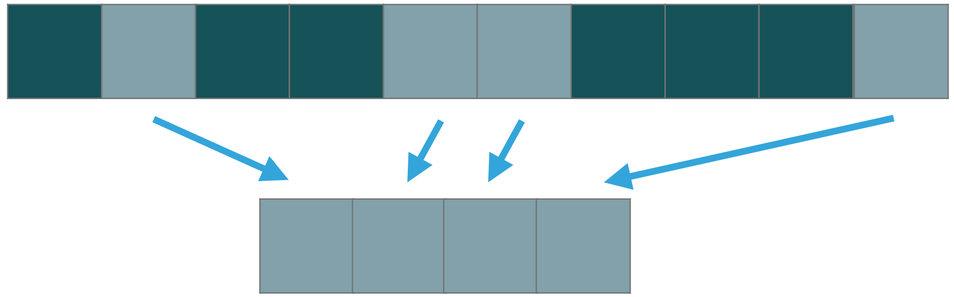
FILTERS

`views::remove_if`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::remove_if(rng, [](int i) { return i % 2 == 0; }),
  {1, 3, 5, 7, 9});
```

FILTERS

`views::remove_if`



```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::remove_if(rng, [](int i) { return i % 2 == 0; }),
  {1, 3, 5, 7, 9});
```

FILTERS

views::take

views::take_exactly

views::take_while

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::take(rng, 6), {0, 1, 2, 3, 4, 5});
check_equal(views::take(rng, 12), rng);

check_equal(views::take_exactly(rng, 6), {0, 1, 2, 3, 4, 5});
// check_equal(views::take_exactly(rng, 12), rng); UB!!

check_equal(views::take_while(rng, [](int i) { return i < 6; }),
  {0, 1, 2, 3, 4, 5});
```

FILTERS

views::drop

views::drop_exactly

views::drop_while

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::drop(views::all(rng), 6), {6, 7, 8, 9, 10}); }
assert(empty(views::drop(views::all(rng), 12)));

check_equal(views::drop_exactly(views::all(rng), 6), {6, 7, 8, 9, 10}); }
// views::drop_exactly(views::all(rng), 12); UB!!

check_equal(views::drop_while(rng, [](int i) { return i < 6; }),
 {6, 7, 8, 9, 10});
```

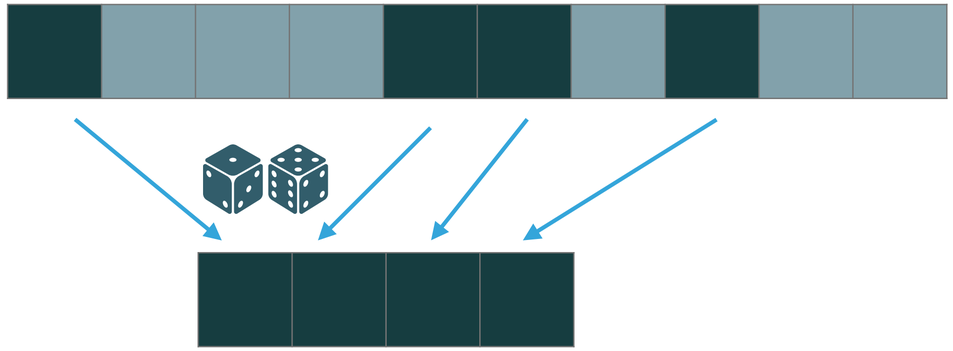
FILTERS

`views::delimit`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
check_equal(views::delimit(rng, 5), {0, 1, 2, 3, 4});
```

FILTERS

`views::sample`



```
auto rng = views::closed_iota('a', 'z');
assert(distance(views::sample(rng, 5)) == 5);
assert(is_sorted(views::sample(rng, 5)));
```

FILTERS

`views::slice`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6};
check_equal(views::slice(rng, 2, 5), {2, 3, 4});
check_equal(views::slice(rng, 2, end - 2), {2, 3, 4});
```

FILTERS

`views::stride`

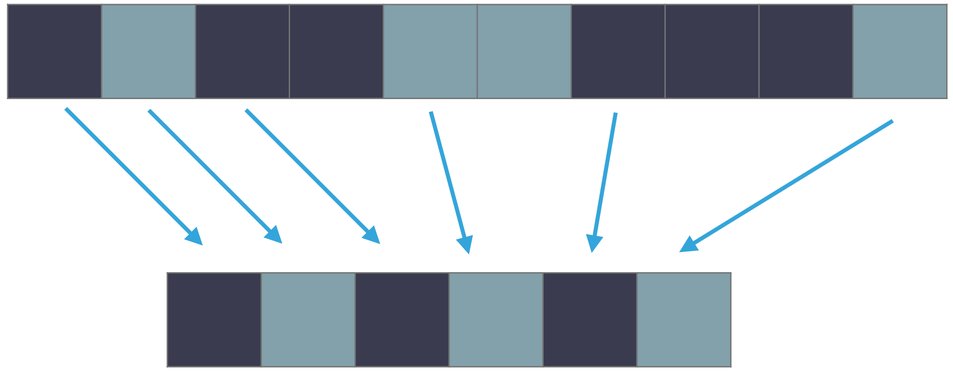
```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::stride(rng, 3), {0, 3, 6, 9});
```

FILTERS

`views::unique`

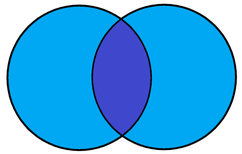
`views::adjacent_remove_if`

`views::adjacent_filter`

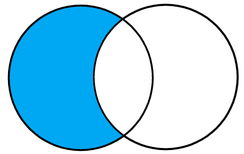


```
const int rng[] = {1, 1, 1, 2, 5, 5, 5, 3};
check_equal(views::unique(rng), {1, 2, 5, 3});
check_equal(views::adjacent_remove_if(rng, ranges::equal_to{}),
  {1, 2, 5, 3});
check_equal(views::adjacent_filter(rng, ranges::not_equal_to{}),
  {1, 2, 5, 3});
```

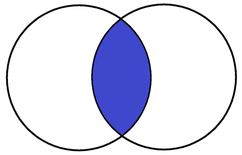
SET VIEWS



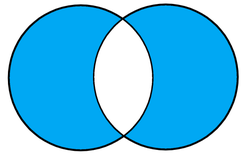
views::set_union



views::set_difference



views::set_intersection



views::set_symmetric_difference

```
auto multiples_of_3 = views::stride(views::ints, 3);
auto squares = views::transform(views::ints, [](int x) { return x * x; });

check_equal(views::take(views::set_difference(multiples_of_3, squares), 6),
  {3, 6, 12, 15, 18, 21});
check_equal(views::take(views::set_intersection(multiples_of_3, squares), 6),
  {0, 9, 36, 81, 144, 225});
check_equal(views::take(views::set_union(multiples_of_3, squares), 6),
  {0, 1, 3, 4, 6, 9});
check_equal(views::take(views::set_symmetric_difference(multiples_of_3, squares), 6),
  {1, 3, 4, 6, 12, 15});
```

TO MULTIPLE RANGES

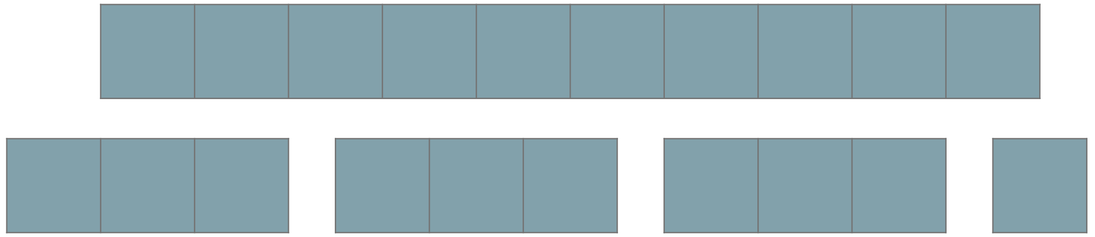
TO MULTIPLE RANGES

`views::chunk`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
auto &&res = views::chunk(rng, 3);
assert(size(res) == 4);
check_equal(res[0], {0, 1, 2});
check_equal(res[1], {3, 4, 5});
check_equal(res[2], {6, 7, 8});
check_equal(res[3], {9});
```

TO MULTIPLE RANGES

`views::chunk`



```
const int rng[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
auto &&res = views::chunk(rng, 3);
assert(size(res) == 4);
check_equal(res[0], {0, 1, 2});
check_equal(res[1], {3, 4, 5});
check_equal(res[2], {6, 7, 8});
check_equal(res[3], {9});
```

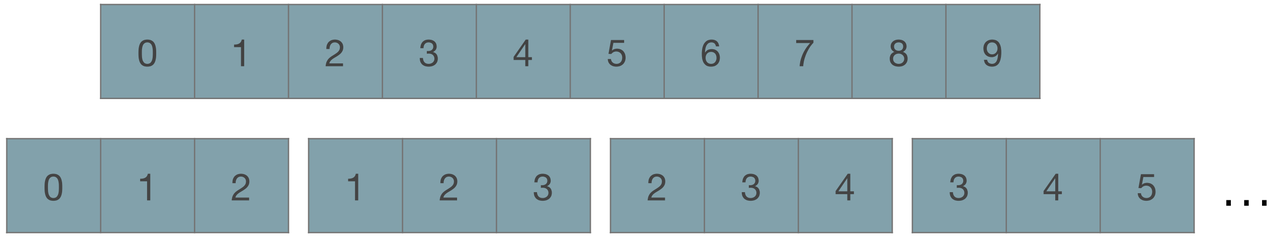
TO MULTIPLE RANGES

`views::sliding`


```
const int rng[] = {0, 1, 2, 3, 4, 5, 6};
auto &&res = views::sliding(rng, 3);
assert(distance(res) == 5);
for (auto &&i : views::indices(5)) {
  check_equal(res[i], {i, i + 1, i + 2});
}
```

TO MULTIPLE RANGES

`views::sliding`




```
const int rng[] = {0, 1, 2, 3, 4, 5, 6};
auto &&res = views::sliding(rng, 3);
assert(distance(res) == 5);
for (auto &&i : views::indices(5)) {
  check_equal(res[i], {i, i + 1, i + 2});
}
```

TO MULTIPLE RANGES

views::split

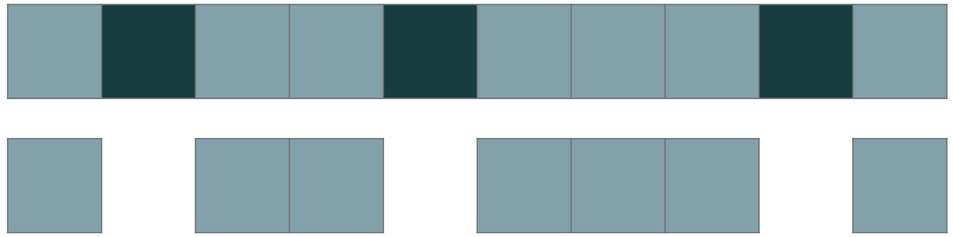
views::split_when

```
const int rng[] = {0, 1, 2, 0, 1, 3, 0, 1, 4};
auto &&res = views::split(rng, 0);
assert(distance(res) == 4);
check_equal(*next(begin(res), 0), views::empty<int>);
check_equal(*next(begin(res), 1), {1, 2});
check_equal(*next(begin(res), 2), {1, 3});
check_equal(*next(begin(res), 3), {1, 4});
```

TO MULTIPLE RANGES

`views::split`

`views::split_when`



```
const int rng[] = {0, 1, 2, 0, 1, 3, 0, 1, 4};
auto &&res = views::split(rng, 0);
assert(distance(res) == 4);
check_equal(*next(begin(res), 0), views::empty<int>);
check_equal(*next(begin(res), 1), {1, 2});
check_equal(*next(begin(res), 2), {1, 3});
check_equal(*next(begin(res), 3), {1, 4});
```

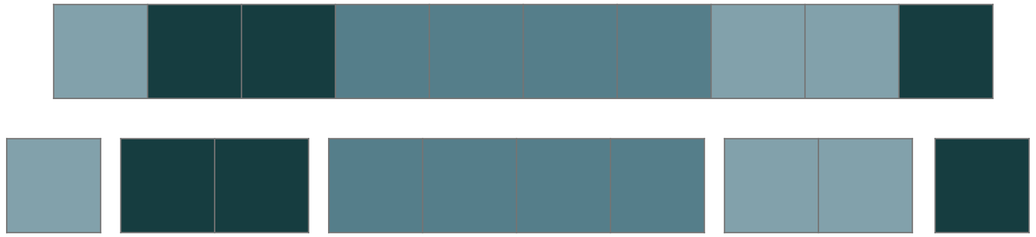
TO MULTIPLE RANGES

`views::group_by`

```
const int rng[] = {1, 1, 2, 2, 1, 3};
auto &&res = views::group_by(rng, ranges::equal_to{});
assert(distance(res) == 4);
check_equal(*next(begin(res), 0), {1, 1});
check_equal(*next(begin(res), 1), {2, 2});
check_equal(*next(begin(res), 2), {1});
check_equal(*next(begin(res), 4), {3});
```

TO MULTIPLE RANGES

`views::group_by`



```
const int rng[] = {1, 1, 2, 2, 1, 3};
auto &&res = views::group_by(rng, ranges::equal_to{});
assert(distance(res) == 4);
check_equal(*next(begin(res), 0), {1, 1});
check_equal(*next(begin(res), 1), {2, 2});
check_equal(*next(begin(res), 2), {1});
check_equal(*next(begin(res), 4), {3});
```

FROM MULTIPLE RANGES

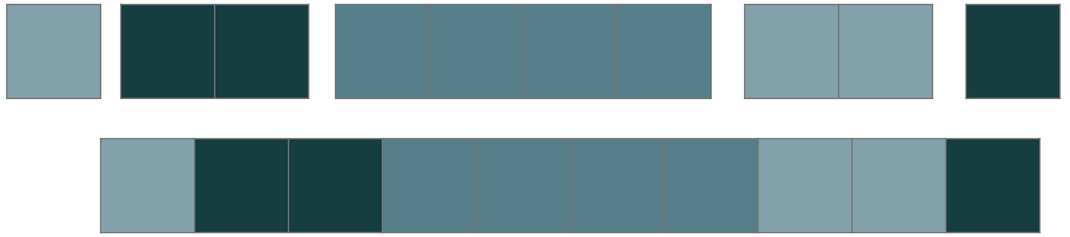
FROM MULTIPLE RANGES

`views::concat`

```
const int rng0[] = {1, 2, 3}, rng1[] = {4, 5, 6}, rng2[] = {7, 8};
check_equal(views::concat(rng0, rng1, rng2),
  {1, 2, 3, 4, 5, 6, 7, 8});
```

FROM MULTIPLE RANGES

`views::concat`



```
const int rng0[] = {1, 2, 3}, rng1[] = {4, 5, 6}, rng2[] = {7, 8};
check_equal(views::concat(rng0, rng1, rng2),
  {1, 2, 3, 4, 5, 6, 7, 8});
```

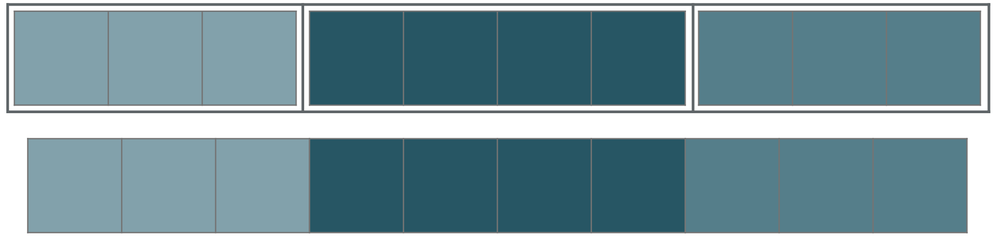
FROM MULTIPLE RANGES

`views::join`

```
const std::vector<std::vector<int>> rng{{0, 1, 2},
  {3, 4, 5, 6},
  {7, 8, 9}};
check_equal(views::join(rng), {0, 1, 2, 3, 4, 5, 6, 7, 8, 9});
```

FROM MULTIPLE RANGES

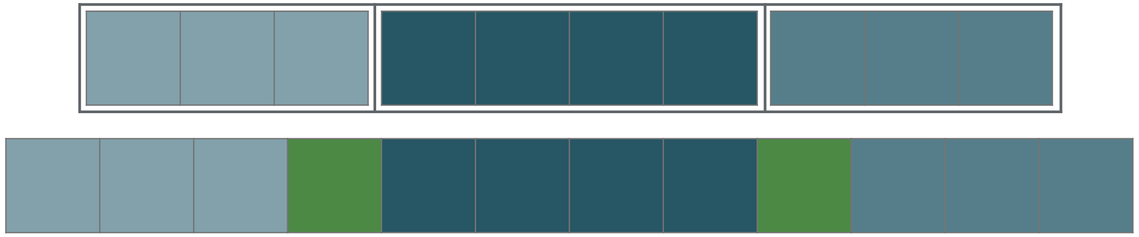
`views::join`



```
const std::vector<std::vector<int>> rng{{0, 1, 2},
  {3, 4, 5, 6},
  {7, 8, 9}};
check_equal(views::join(rng), {0, 1, 2, 3, 4, 5, 6, 7, 8, 9});
```

FROM MULTIPLE RANGES

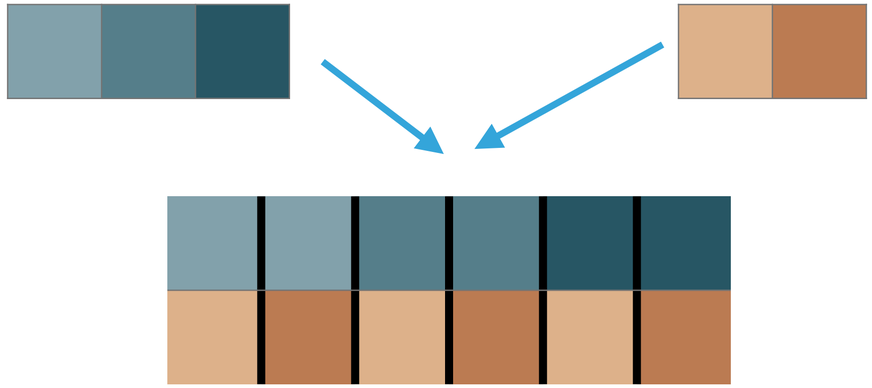
`views::join`



```
const std::vector<std::vector<int>> rng{{0, 1, 2},
  {3, 4, 5, 6},
  {7, 8, 9}};
check_equal(views::join(rng, 42),
  {0, 1, 2, 42, 3, 4, 5, 6, 42, 7, 8, 9});
check_equal(views::join(rng, views::iota(42, 44)),
  {0, 1, 2, 42, 43, 3, 4, 5, 6, 42, 43, 7, 8, 9});
```

FROM MULTIPLE RANGES

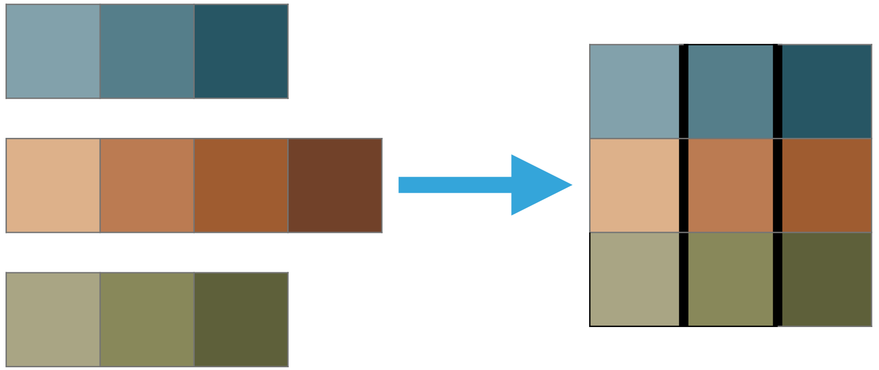
`views::cartesian_product`



```
const int numbers[] = {1, 2, 3};
std::string const names[] = {"eric"s, "casey"s};
check_equal(views::cartesian_product(numbers, names),
{tuple{1, "eric"s}, tuple{1, "casey"s},
tuple{2, "eric"s}, tuple{2, "casey"s},
tuple{3, "eric"s}, tuple{3, "casey"s}});
```

FROM MULTIPLE RANGES

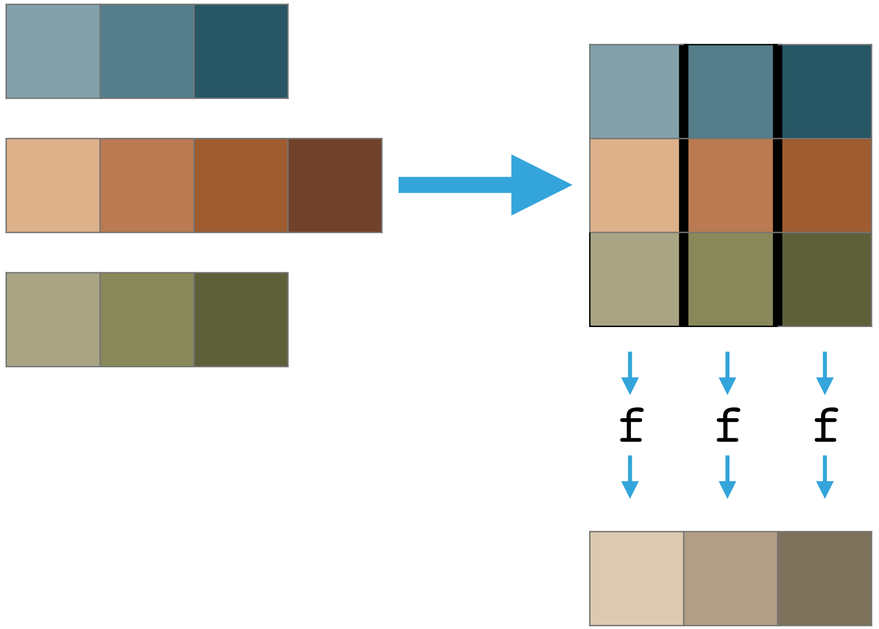
`views::zip`



```
const std::string names[] = {"john", "paul", "george", "richard"};
const int songs[] = {72, 70, 22, 2};
check_equal(views::zip(names, songs), {
  pair{"john"s, 72},
  pair{"paul"s, 70},
  pair{"george"s, 22},
  pair{"richard"s, 2}});
```

FROM MULTIPLE RANGES

`views::zip_with`




```
const std::string names[] = {"john", "paul", "george", "richard"};
const int songs[] = {72, 70, 22, 2};
check_equal(views::zip_with([](const std::string &name, const int songs){
  return name + " wrote " + std::to_string(songs);
}, names, songs),
{"john wrote 72"s, "paul wrote 70"s,
  "george wrote 22"s, "richard wrote 2"s});
```

TRANSFORMERS

TRANSFORMERS

`views::reverse`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6};
check_equal(views::reverse(rng), {6, 5, 4, 3, 2, 1, 0});
```

TRANSFORMERS

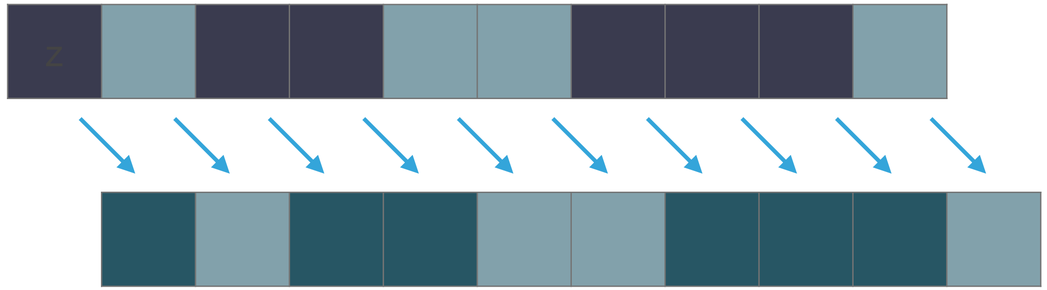
`views::reverse`

```
const int rng[] = {0, 1, 2, 3, 4, 5, 6};
check_equal(views::reverse(rng), {6, 5, 4, 3, 2, 1, 0});
```

TRANSFORMERS

`views::replace`

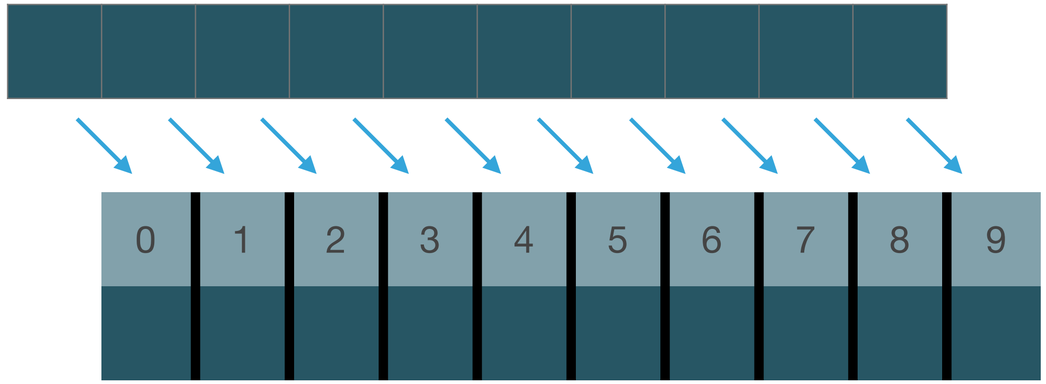
`views::replace_if`



```
const int rng[] = {1, 2, 3, 1, 2, 3, 1, 2, 3};
check_equal(views::replace(rng, 1, 42),
  {42, 2, 3, 42, 2, 3, 42, 2, 3});

check_equal(views::replace_if(rng, [](int i) {
  return i != 1;
}, 42),
  {1, 42, 42, 1, 42, 42, 1, 42, 42});
```

TRANSFORMERS

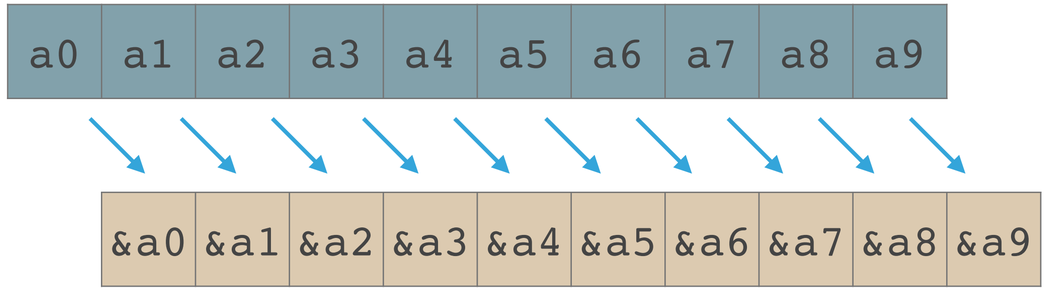
`views::enumerate`




```
std::string const names[] = {"adi"s, "michael"s, "eran"s, "amir"s};
check_equal(views::enumerate(names), {
  pair{0, "adi"s},
  pair{1, "michael"s},
  pair{2, "eran"s},
  pair{3, "amir"s}});
```

TRANSFORMERS

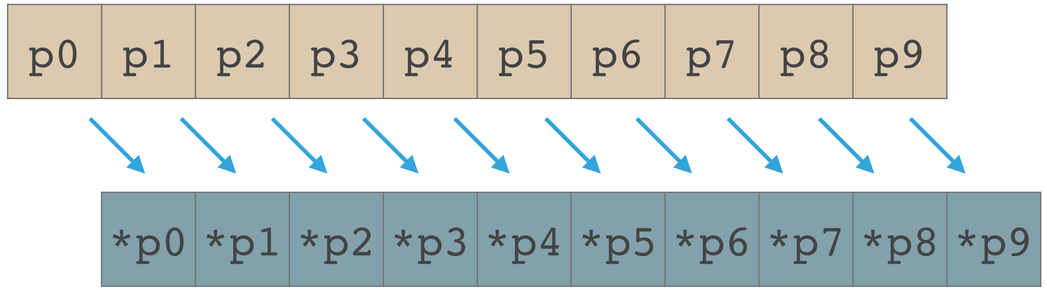
`views::addressof`




```
const int rng[] = {1, 2, 3, 4};
check_equal(views::addressof(rng),
  {rng, rng + 1, rng + 2, rng + 3});
```

TRANSFORMERS

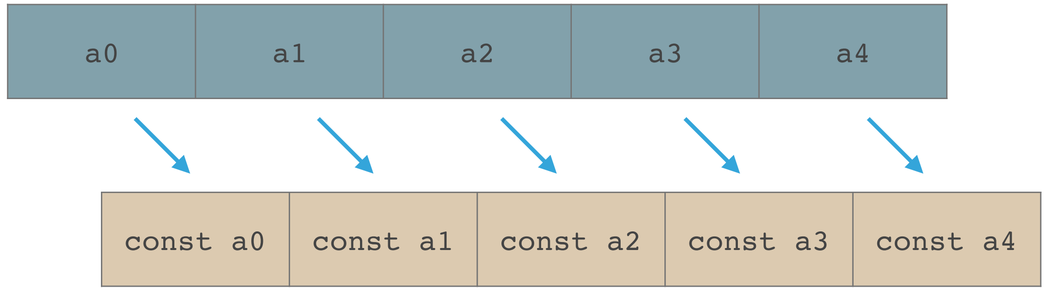
`views::indirect`




```
const int arr[] = {0, 1, 2, 3, 4};
auto &&rng = views::iota(arr, arr + 5);
check_equal(views::indirect(rng), arr);
```

TRANSFORMERS

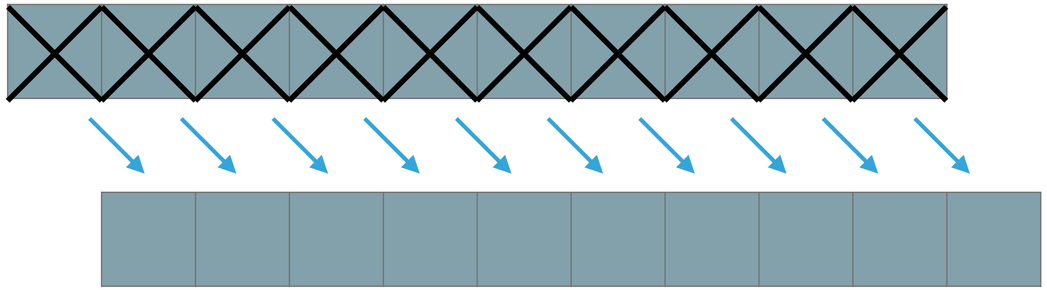
`views::const_`




```
int rng[] = {1, 2, 3, 4};
auto &&res = views::const_(rng); static_assert(
  std::is_same_v<range_reference_t<decltype(rng)>, int &>,
  "default should be non const"); static_assert(
  std::is_same_v<range_reference_t<decltype(res)>, const int &>,
  "const_ should be const");
check_equal(res, {1, 2, 3, 4});
```

TRANSFORMERS

`views::move`



```
std::vector<std::vector<int>> source{
  {0, 1, 2}, {3, 4, 5, 6}, {7, 8, 9}};
std::vector<std::vector<int>> dest{3};
copy(views::move(source), begin(dest));
check_equal(dest,
  std::vector<std::vector<int>>{{0, 1, 2}, {3, 4, 5, 6}, {7, 8, 9}});
check_equal(source, std::vector<std::vector<int>>{3});
```

TRANSFORMERS

`views::elements`

```
std::map<int, std::string> m{{1, "one"}, {2, "two"}, {3, "three"}};
check_equal(views::elements<0>(m), {1, 2, 3});
check_equal(views::elements<1>(m), {"one"s, "two"s, "three"s});
```

TRANSFORMERS

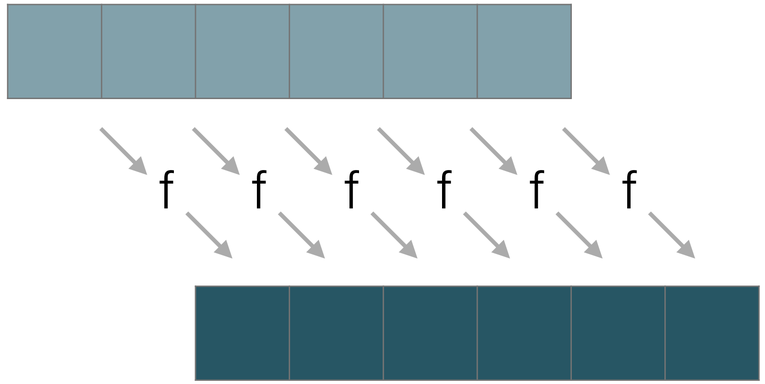
- `views::elements`

```
std::map<int, std::string> m{{1, "one"}, {2, "two"}, {3, "three"}};
check_equal(views::elements<0>(m), {1, 2, 3});
check_equal(views::elements<1>(m), {"one"s, "two"s, "three"s});
```

- `views::keys == views::elements<0>`
- `views::values == views::elements<1>`

TRANSFORMERS

`views::transform`




```
const int rng[] = {0, 1, 2, 3, 4, 5};
check_equal(views::transform(rng, [](int i) {
  return std::to_string(i);
}),
  {"0"s, "1"s, "2"s, "3"s, "4"s, "5"s});
```

ADDING ELEMENTS

ADDING ELEMENTS

`views::intersperse`

```
const int rng[] = {0, 1, 2, 3, 4};
check_equal(views::intersperse(rng, 42),
  {0, 42, 1, 42, 2, 42, 3, 42, 4});
```

ADDING ELEMENTS

`views::intersperse`

```
const int rng[] = {0, 1, 2, 3, 4};
check_equal(views::intersperse(rng, 42),
  {0, 42, 1, 42, 2, 42, 3, 42, 4});
```

ADDING ELEMENTS

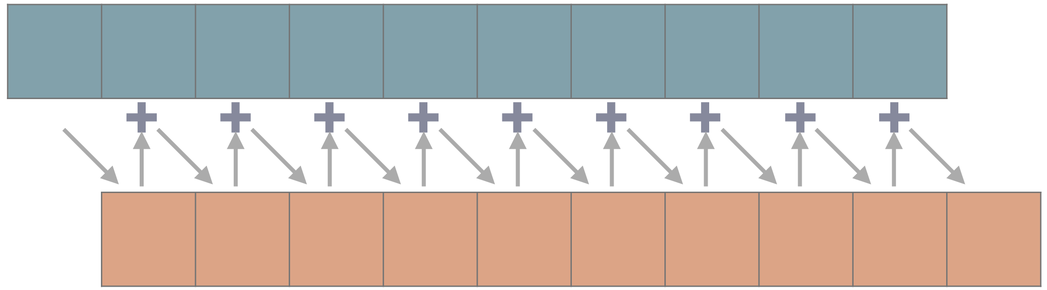
`views::cycle`

...

```
const int rng[] = {0, 1, 2};
check_equal(views::take(views::cycle(rng), 10),
  {0, 1, 2, 0, 1, 2, 0, 1, 2, 0});
```

NUMERICS

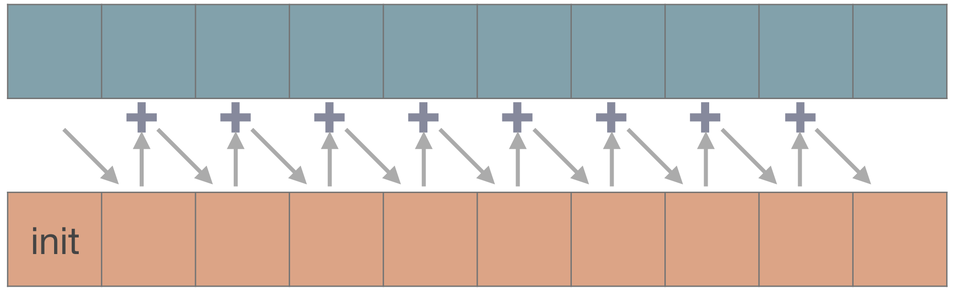
`views::partial_sum`



```
const int rng[] = {0, 1, 2, 3, 4, 5, 6};
auto &&res = views::partial_sum(rng, plus{});
check_equal(res, {0, 1, 3, 6, 10, 15, 21});
```

NUMERICS

`views::exclusive_scan`




```
const int rng[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
check_equal(views::exclusive_scan(rng, 0),
  {0, 1, 3, 6, 10, 15, 21, 28, 36, 45});
```

UTILITIES

`views::all`

`range` → `view`

```
const int rng[] = {1, 2, 3, 4};
check_equal(views::all(rng), {1, 2, 3, 4});
```

UTILITIES

`views::common`


```
auto &&res = views::common(views::iota(0, 4));
assert(std::equal(res.begin(), res.end(), begin(views::iota(0, 4))));
```

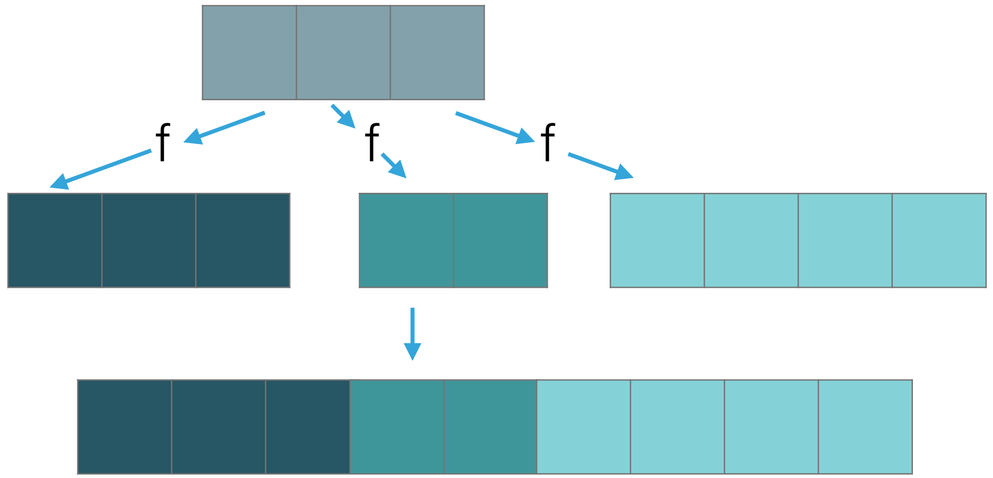
UTILITIES

`any_view<T, Cat = category::input>` Can store any view of category Cat having value type **`T`**

```
any_view<int, category::forward> any;
assert(empty(any));
any = views::iota(0, 4);
any = views::single(42);
check_equal(any, {42});
// any = views::generate_n([](){ return 42; }, 4); does not compile
```

LIST COMPREHENSION

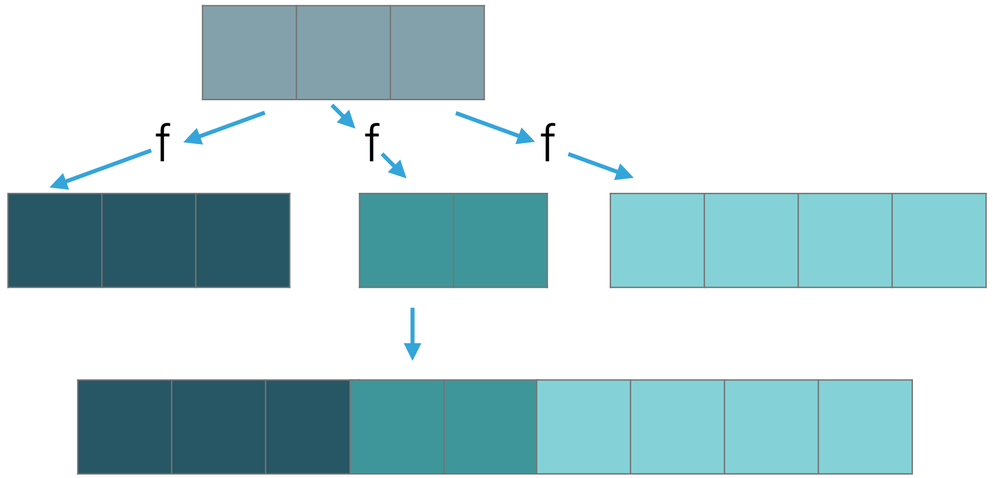
`views::for_each`




```
const int rng[] = {0, 1, 2, 3};
check_equal(views::for_each(rng, [](int i) {
  return yield(i * i);
}),
  {0, 1, 4, 9});
```

LIST COMPREHENSION

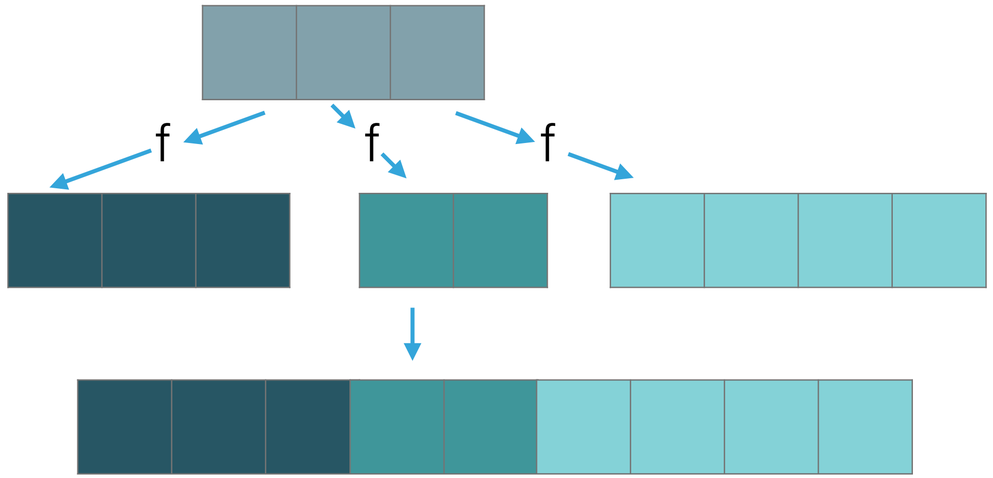
`views::for_each`




```
const int rng[] = {0, 1, 2, 3};
check_equal(views::for_each(rng, [](int i) {
  return yield_from(views::indices(i));
}),
  {0, 0, 1, 0, 1, 2});
```

LIST COMPREHENSION

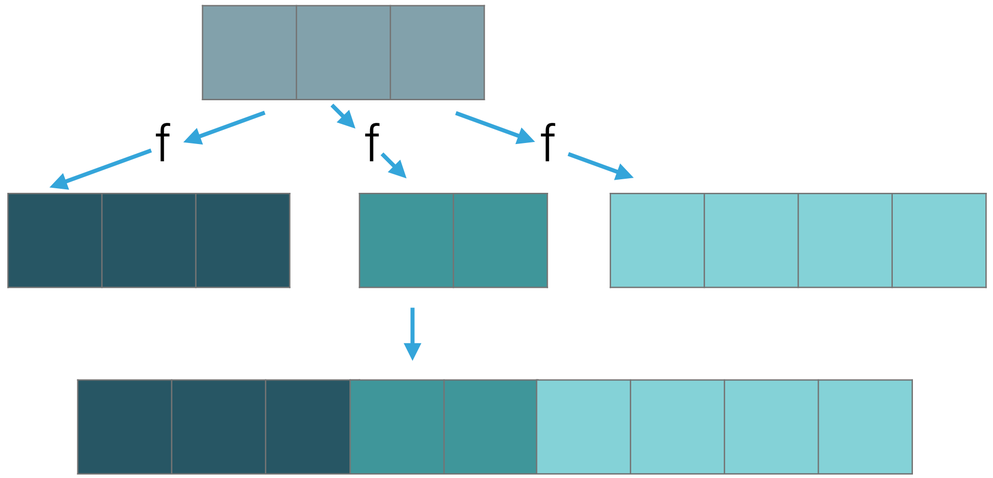
`views::for_each`




```
const int rng[] = {0, 1, 2, 3};
check_equal(views::for_each(rng, [](int i) {
  return yield_if(i % 2 == 0, i / 2);
}),
  {0, 1});
```

LIST COMPREHENSION

`views::for_each`




```
const int rng[] = {0, 1, 2, 3};
check_equal(views::for_each(rng, [](int i) {
  return lazy_yield_if(i % 2 == 0, [i] {
    return i / 2;
  });
}),
  {0, 1});
```

TO BE ON YOUR OWN, WITH NO

COMPOSITION

Yaacov Simhon

CLASSIC STL

```
int sum_of_squares(int count) {
  std::vector<int> numbers(static_cast<size_t>(count));
  std::iota(numbers.begin(), numbers.end(), 1);
  std::transform(numbers.begin(), numbers.end(), numbers.begin(),
    [](int x) { return x * x; });
  return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```

FUNCTION CALL SYNTAX

```
int sum_of_squares(int count) {
  return accumulate(
    views::transform(
      views::iota(1, count),
      [](int x) { return x * x; }
    ), 0
  );
}
```

PIPED SYNTAX

```
int sum_of_squares(int count) {
  auto squares = views::iota(1, count)
               | views::transform([](int x) { return x * x; });
  return accumulate(squares, 0);
}
```

VIEW MATERIALIZATION

```
auto vec = views::ints
  | views::transform([](int i) { return i + 42; })
  | views::take(10)
  | to<std::vector>;

static_assert(std::is_same_v<decltype(vec), std::vector<int>);
```

WON'T GET FOOLED

ACTIONS

COMPLETING THE PICTURE

- algorithms - eager but don’t compose
- views - lazy and compose
- What about composable AND eager?
- Enter actions…

ADDING ELEMENTS

`actions::push_back`

```
std::vector rng{0, 1, 2};

check_equal(actions::push_back(rng, 42), {0, 1, 2, 42});

check_equal(actions::push_back(rng, views::indices(40, 42)),
  {0, 1, 2, 40, 41});
```

ADDING ELEMENTS

`actions::push_front`

```
std::list rng{0, 1, 2};

check_equal(actions::push_front(rng, 42), {42, 0, 1, 2});

check_equal(actions::push_front(rng, views::indices(40, 42)),
  {40, 41, 0, 1, 2});
```

ADDING ELEMENTS

`actions::insert`

```
std::set<int> rng;

auto &&[it, inserted] = actions::insert(rng, 42);
assert(inserted);
assert(*it == 42);
check_equal(rng, {42});

actions::insert(rng, views::indices(5));
check_equal(rng, {0, 1, 2, 3, 4, 42});
```

ADDING ELEMENTS

`actions::insert`

```
std::set<int> rng;

auto &&[it, inserted] = actions::insert(rng, 42);
assert(inserted);
assert(*it == 42);
check_equal(rng, {42});

actions::insert(rng, views::indices(5));
check_equal(rng, {0, 1, 2, 3, 4, 42});
```

REMOVING ELEMENTS

`actions::erase`

```
std::vector rng{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto &&res = actions::erase(rng, begin(rng), next(begin(rng), 2));
assert(*res == 2);
check_equal(rng, {2, 3, 4, 5, 6, 7, 8, 9, 10});
```

REMOVING ELEMENTS

`actions::erase`

```
std::vector rng{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto &&res = actions::erase(rng, begin(rng), next(begin(rng), 2));
assert(*res == 2);
check_equal(rng, {2, 3, 4, 5, 6, 7, 8, 9, 10});
```

PERMUTATIONS

`actions::sort`

`actions::stable_sort`

```
std::vector rng{"Jeff"s, "Bill"s, "Warren"s, "Bernard"s, "Carlos"s};
check_equal(actions::sort(rng),
  {"Bernard"s, "Bill"s, "Carlos"s, "Jeff"s, "Warren"s});
```

PERMUTATIONS

`actions::sort`

`actions::stable_sort`

```
std::vector rng{"Jeff"s, "Bill"s, "Warren"s, "Bernard"s, "Carlos"s};
check_equal(actions::sort(rng),
  {"Bernard"s, "Bill"s, "Carlos"s, "Jeff"s, "Warren"s});
```

PERMUTATIONS

`actions::shuffle`

```
std::vector rng{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
assert(!is_sorted(actions::shuffle(rng, std::mt19937{})));
```

PERMUTATIONS

`actions::shuffle`

```
std::vector rng{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
assert(!is_sorted(actions::shuffle(rng, std::mt19937{})));
```

AND MORE...

- `actions::take`
- `actions::take_while`
- `actions::drop`
- `actions::drop_while`
- `actions::remove_if`
- `actions::unique`
- `actions::slice`
- `actions::stride`
- `actions::reverse`
- `actions::transform`
- `actions::split`

COMPOSABLE

```
extern std::vector<int> read_data();

auto vi = read_data()
        | actions::sort
        | actions::unique;

static_assert(std::is_same_v<decltype(vi), std::vector<int>>);
```

NOTE

- Pipelines work on rvalue containers.
- For lvalues:

```
auto v2 = v | copy | actions::sort;
auto v3 = v | move | actions::sort;
```

- Shortcut:

```
v |= actions::sort;
```

Photo by (c)Lior Keter

I'LL JUST END UP WALKIN'
IN THE COLD

PROJECTIONS

ranges::for_each

```
namespace ranges {

  template<input_iterator I, sentinel_for<I> S, class Proj = identity,
           indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each(I first, S last, Fun f, Proj proj = {});

  template<input_range R, class Proj = identity,
           indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
      for_each(R&& r, Fun f, Proj proj = {});
}
```

MOTIVATION

```
struct employee {
  std::string first_name;
  std::string last_name;
};

std::vector<employee> employees;

std::sort(employees.begin(), employees.end(),
  [](const employee &x, const employee &y) {
    return x.last_name < y.last_name;
});

auto p = std::lower_bound(employees.begin(), employees.end(), "Niebler",
  [](const employee &x, const std::string &y) {
    return x.last_name < y;
});
```

PROJECT

```
template<input_iterator I, sentinel_for<I> S, class Proj = identity,
         indirectly_unary_invocable<projected<I, Proj>> Fun>
  constexpr for_each_result<I, Fun>
    for_each(I first, S last, Fun f, Proj proj = {}) {

      for (; first != last; ++first) {
        invoke(fun, invoke(proj, *first));
      }

      return {std::move(first), std::move(fun)};
}
```

USAGE

```
struct employee {
  std::string first_name;
  std::string last_name;
};

std::vector<employee> employees;

auto get_last_name = [](const employee &e) { return e.last_name; };


sort(employees, less{}, get_last_name);
auto p = lower_bound(employees, "Niebler", less{}, get_last_name);
```

AUTO MEMBER PROJECTION

```
struct employee {
  std::string first_name;
  std::string last_name;
};

std::vector<employee> employees;
```

```
sort(employees, {}, &employee::last_name);
auto p = lower_bound(employees, "Niebler", {}, &employee::last_name);
```

WE'LL GIVE YOU

(A TOUCH OF) PERFORMANCE

CLASSIC STL

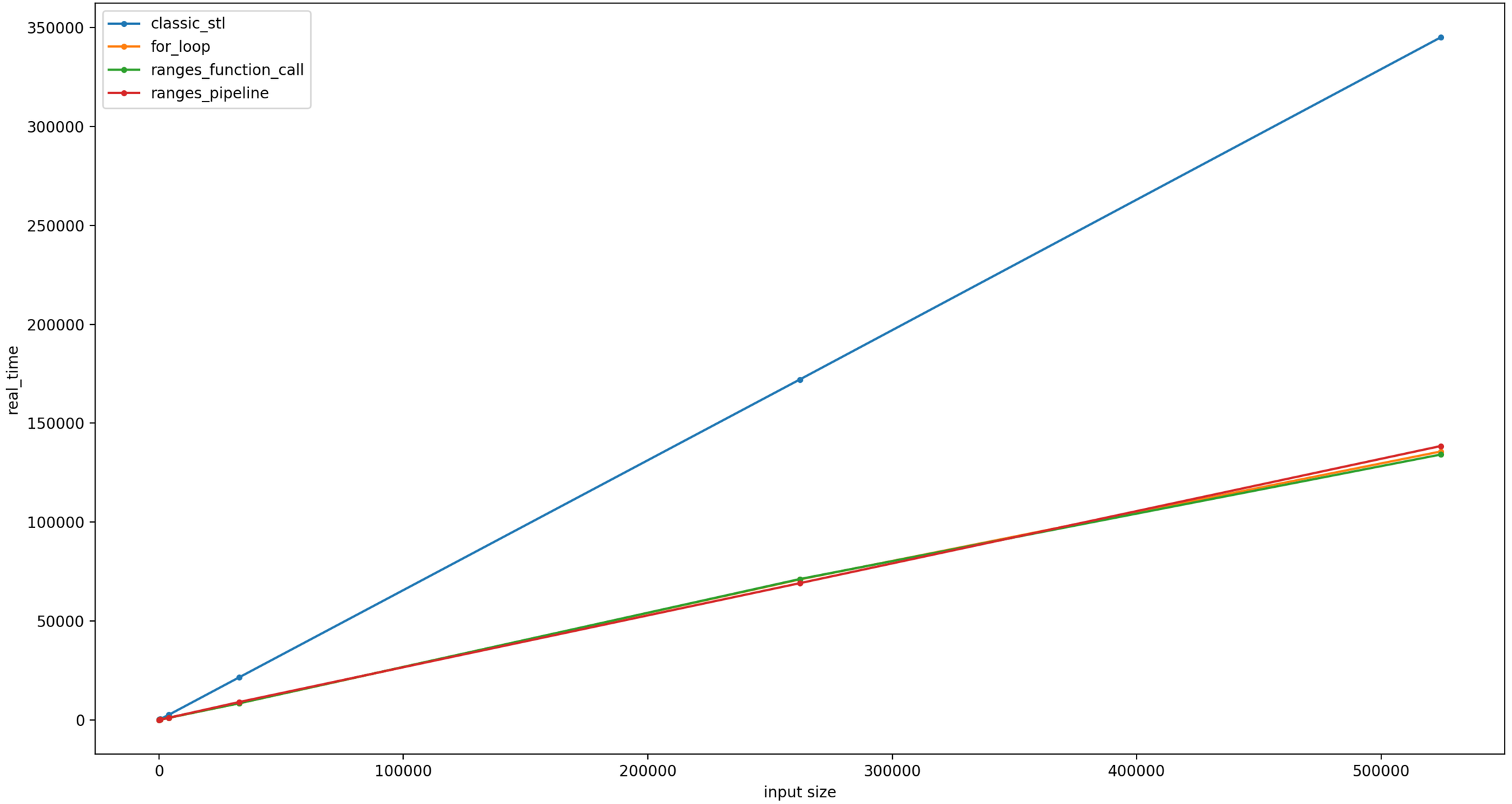
```
int sum_of_squares(int count) {
  std::vector<int> numbers(static_cast<size_t>(count));
  std::iota(numbers.begin(), numbers.end(), 1);
  std::transform(numbers.begin(), numbers.end(), numbers.begin(),
    [](int x) { return x * x; });
  return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```

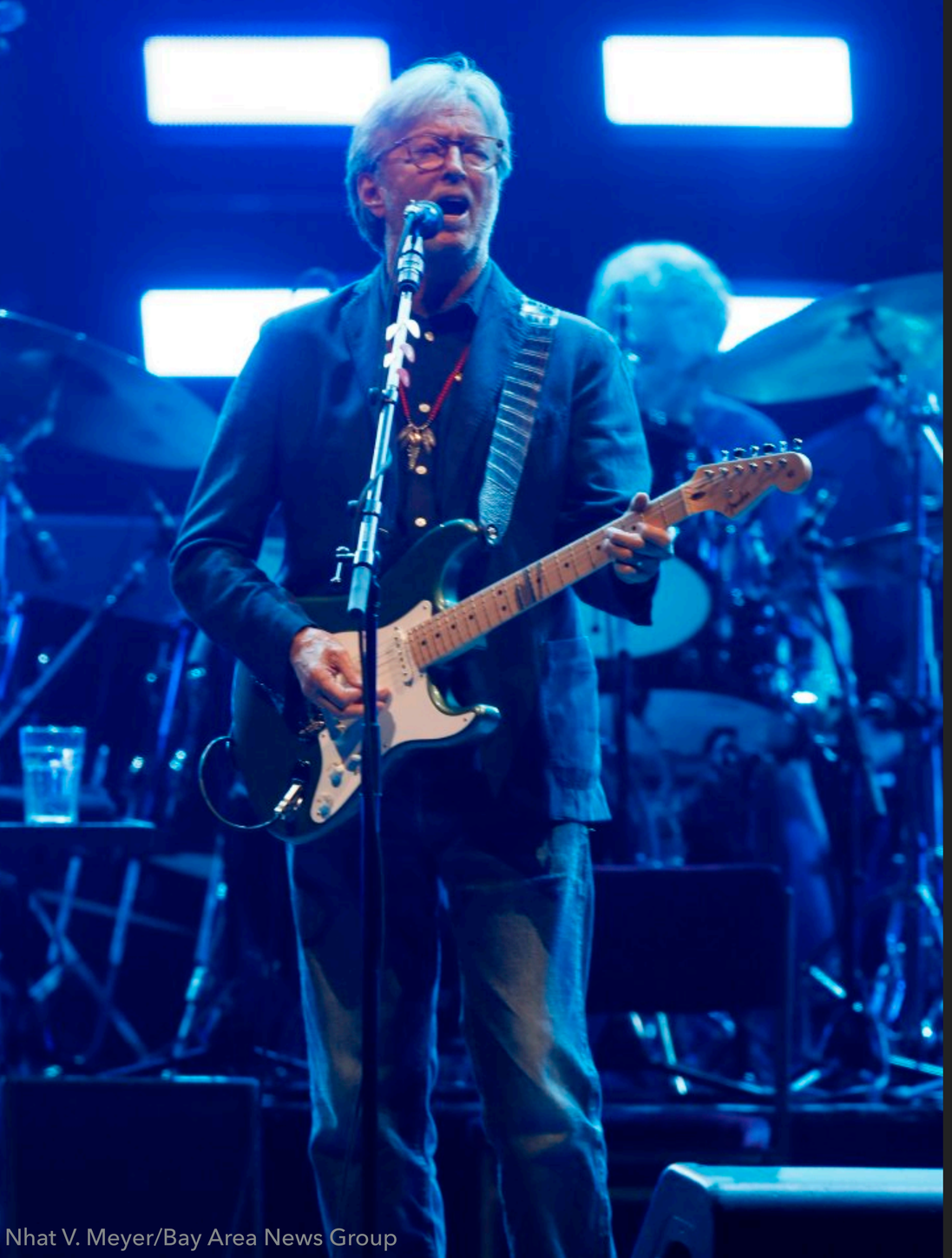
FUNCTION CALL SYNTAX

```
int sum_of_squares(int count) {
  return accumulate(
    views::transform(
      views::iota(1, count),
      [](int x) { return x * x; }
    ), 0
  );
}
```

PIPED SYNTAX

```
int sum_of_squares(int count) {
  auto squares = views::iota(1, count)
               | views::transform([](int x) { return x * x; });
  return accumulate(squares, 0);
}
```


classic_stl
for_loop
ranges_function_call
ranges_pipeline
real_time
input size
0
50000
100000
150000
200000
250000
300000
350000
0
100000
200000
300000
400000
500000



Nhat V. Meyer/Bay Area News Group

https://github.com/dvirtz/ranges_code_samples

@dvirtzwastaken

I WANT TO

THANK YOU

https://github.com/dvirtz/ranges_code_samples

@dvirtzwastaken

LOVE FIRST, ASK

QUESTIONS

LATER