Ranges adaptors for non-copyable iterators

Document #:	P1862R1
Date:	2019-11-08
Project:	Programming Language C++
Audience:	LEWG, LWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	Casey Carter <Casey@Carter.net>

Introduction

This paper is an addendum to P1207 and add support where possible to the iterators of the views introduced by The one range proposal as well as P1035, with the apologies of the author for the tardiness of that work.

Points of design

Views and adaptors constructed from iterators (whether that is two iterators, an iterator and an index or an iterator and a sentinel) need to store these iterators - which would subsequently need to be copied by calls to `begin()`.

Note that because iterating over a container does not invalidate the storage, all sensible ranges with non-copyable iterators are views.

All other views support move-only iterators (with the exception of reverse view which require bidirectional iterators).

Caching of move-only iterators is not supported either, which is sensible given single-pass iterators are invalided by reading from the range.

`views::counted` and `subrange`

Unlike other views, `subrange` does store the begin iterator. We therefore have two options in the C++20 time frame:

- Require copyability for `subrange`, and adapt `views::counted` at a later date to be constructed from a range rather than an iterator, therefore bypassing the need for a copy in the `begin` call.

  [*Note:* Neither `subrange` nor `views::counted` are necessary to implement other standard views and algorithms operating on non forward iterators and ranges. — *end note*]

- Move the iterator in the call to `begin()` in `views::subrange`. This is a simple change but requires the adoption of [P1456].

  [*Note:* In the general case, it is not guaranteed that the iterators returned by two successive calls to `begin` will be usable. (In all cases, dereferencing one of these iterator after the other has been incremented is undefined behavior).

  So moving the iterator out if the view is in this case reasonable. — *end note*]

  There is also the question of whether `subrange::begin` should be const for non-copyable iterators. Constness is necessary to satisfy the requirements of *simple-view*, but this would require the iterator to be mutable and to mutate it in a const method... The author prefers `subrange::begin` to be non-const if the iterator is not copyable and so in this case `subrange` would not satisfy the requirements of *simple-view*.

  The provided wording for this second option as approved by LEWG.

Implementation

This proposal has been implemented in cmcstl2.

Acknowledgments

Many thanks to Eric Niebler, Casey Carter, Christopher Di Bella and Tristan Brindle for reviewing this work and providing valuable feedback.

Wording

[ranges.syn] Header `<ranges>` synopsis

```
#include <initializer_list>
#include <iterator>

namespace std::ranges {
```

```
inline namespace unspecified {
        // ??, range access
        inline constexpr unspecified begin = unspecified;
        inline constexpr unspecified end = unspecified;
        inline constexpr unspecified cbegin = unspecified;
        inline constexpr unspecified cend = unspecified;
        inline constexpr unspecified rbegin = unspecified;
        inline constexpr unspecified rend = unspecified;
        inline constexpr unspecified crbegin = unspecified;
        inline constexpr unspecified crend = unspecified;

        inline constexpr unspecified size = unspecified;
        inline constexpr unspecified empty = unspecified;
        inline constexpr unspecified data = unspecified;
        inline constexpr unspecified cdata = unspecified;
}

// ??, ranges
template<class T>
concept range = see below;

template<range R>
using iterator_t = decltype(ranges::begin(declval<R&>()));
template<range R>
using sentinel_t = decltype(ranges::end(declval<R&>()));
template<range R>
using range_difference_t = iter_difference_t<iterator_t<R>>;
template<range R>
using range_value_t = iter_value_t<iterator_t<R>>;
template<range R>
using range_reference_t = iter_reference_t<iterator_t<R>>;
template<range R>
using range_rvalue_reference_t = iter_rvalue_reference_t<iterator_t<R>>;

// ??, sized ranges
template<class>
inline constexpr bool disable_sized_range = false;

template<class T>
concept sized_range = see below;

// ??, views
template<class T>
inline constexpr bool enable_view = see below;

struct view_base { };

template<class T>
concept view = see below;

// ??, other range refinements
```

```
template<class R, class T>
concept output_range = see below;

template<class T>
concept input_range = see below;

template<class T>
concept forward_range = see below;

template<class T>
concept bidirectional_range = see below;

template<class T>
concept random_access_range = see below;

template<class T>
concept contiguous_range = see below;

template<class T>
concept common_range = see below;

template<class T>
concept viewable_range = see below;

// ??, class template view_interface
template<class D>
requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface;

// ??, sub-ranges
enum class subrange_kind : bool { unsized, sized };

template<input_or_output_iterator I, sentinel_for<I> S = I, subrange_kind K = see below>
requires (K == subrange_kind::sized || !sized_sentinel_for<S, I>)
class subrange;

// ??, dangling iterator handling
struct dangling;

template<range R>
using safe_iterator_t = conditional_t<forwarding-range<R>, iterator_t<R>, dangling>;

template<range R>
using safe_subrange_t =
conditional_t<forwarding-range<R>, subrange<iterator_t<R>>, dangling>;

// ??, empty view
template<class T>
requires is_object_v<T>
class empty_view;
```

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

// ??, single view
template<copy_constructible T>
requires is_object_v<T>
class single_view;

namespace views { inline constexpr unspecified single = unspecified; }

// ??, iota view
template<weakly_incrementable W, semiregular Bound = unreachable_sentinel_t>
requires weakly-equality-comparable-with<W, Bound>
class iota_view;

namespace views { inline constexpr unspecified iota = unspecified; }

// ??, all view
namespace views { inline constexpr unspecified all = unspecified; }

template<viewable_range R>
using all_view = decltype(views::all(declval<R>()));

template<range R>
requires is_object_v<R>
class ref_view;

// ??, filter view
template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
requires view<V> && is_object_v<Pred>
class filter_view;

namespace views { inline constexpr unspecified filter = unspecified; }

// ??, transform view
template<input_range V, copy_constructible F>
requires view<V> && is_object_v<F> &&
regular_invocable<F&, range_reference_t<V>>
class transform_view;

namespace views { inline constexpr unspecified transform = unspecified; }

// ??, take view
template<view> class take_view;

namespace views { inline constexpr unspecified take = unspecified; }

// ??, take while view
template<view R, class Pred>
```

```
requires input_range<R> && is_object_v<Pred> &&
indirect_unary_predicate<const Pred, iterator_t<R>>
class take_while_view;

namespace views { inline constexpr unspecified take_while = unspecified; }

// ??, drop view
template<view R>
class drop_view;

namespace views { inline constexpr unspecified drop = unspecified; }

// ??, drop while view
template<view R, class Pred>
requires input_range<R> && is_object_v<Pred> &&
indirect_unary_predicate<const Pred, iterator_t<R>>
class drop_while_view;

namespace views { inline constexpr unspecified drop_while = unspecified; }

// ??, join view
template<input_range V>
requires view<V> && input_range<range_reference_t<V>> &&
(is_reference_v<range_reference_t<V>> ||
view<range_value_t<V>>)
class join_view;

namespace views { inline constexpr unspecified join = unspecified; }

// ??, split view
template<class R>
concept tiny-range = see below;   // exposition only

template<input_range V, forward_range Pattern>
requires view<V> && view<Pattern> &&
indirectly_comparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
(forward_range<V> || tiny-range<Pattern>)
class split_view;

namespace views { inline constexpr unspecified split = unspecified; }

// ??, counted view
namespace views { inline constexpr unspecified counted = unspecified; }

// ??, common view
template<view V>
requires (!common_range<V> && copyable<iterator_t<V>>)
class common_view;

namespace views { inline constexpr unspecified common = unspecified; }
```

```
        // ??, reverse view
        template<view V>
        requires bidirectional_range<V>
        class reverse_view;

        namespace views { inline constexpr unspecified reverse = unspecified; }

        // ??, istream view
        template<movable Val, class CharT, class Traits = char_traits<CharT>>
        requires see below
        class basic_istream_view;
        template<class Val, class CharT, class Traits>
        basic_istream_view<Val, CharT, Traits> istream_view(basic_istream<CharT, Traits>& s);

        // ??, elements view
        template<input_range R, size_t N>
        requires see below;
        class elements_view;

        template<class R>
        using keys_view = elements_view<all_view<R>, 0>;
        template<class R>
        using values_view = elements_view<all_view<R>, 1>;

        namespace views {
                template<size_t N>
                inline constexpr unspecified elements = unspecified ;
                inline constexpr unspecified keys = unspecified ;
                inline constexpr unspecified values = unspecified ;
        }
}
```

Sub-ranges [range.subrange]

The `subrange` class template combines together an iterator and a sentinel into a single object that models the `view` concept. Additionally, it models the `sized_range` concept when the final template parameter is `subrange_kind::sized`.

```
        namespace std::ranges {
                template<class T>
                concept pair-like =                                    // exposition only
                !is_reference_v<T> && requires(T t) {
                        typename tuple_size<T>::type;   // ensures tuple_size<T> is complete
                        requires derived_from<tuple_size<T>, integral_constant<size_t, 2>>;
                        typename tuple_element_t<0, remove_const_t<T>>;
                        typename tuple_element_t<1, remove_const_t<T>>;
                        { get<0>(t) } -> convertible_to<const tuple_element_t<0, T>&>;
                        { get<1>(t) } -> convertible_to<const tuple_element_t<1, T>&>;
                };

                template<class T, class U, class V>
```

```
        concept pair-like-convertible-to =                    // exposition only
        !range<T> && pair-like<remove_reference_t<T>> &&
        requires(T&& t) {
                { get<0>(std::forward<T>(t)) } -> convertible_to<U>;
                { get<1>(std::forward<T>(t)) } -> convertible_to<V>;
        };

        template<class T, class U, class V>
        concept pair-like-convertible-from =                  // exposition only
        !range<T> && pair-like<T> && constructible_from<T, U, V>;

        template<class T>
        concept iterator-sentinel-pair =                      // exposition only
        !range<T> && pair-like<T> &&
        sentinel_for<tuple_element_t<1, T>, tuple_element_t<0, T>>;

        template<input_or_output_iterator I, sentinel_for<I> S = I, subrange_kind K =
        sized_sentinel_for<S, I> ? subrange_kind::sized : subrange_kind::unsized>
        requires (K == subrange_kind::sized || !sized_sentinel_for<S, I>)
        class subrange : public view_interface<subrange<I, S, K>> {
                private:
                static constexpr bool StoreSize =                     // exposition
only
                K == subrange_kind::sized && !sized_sentinel_for<S, I>;
                I begin_ = I();                                       // exposition
only
                S end_ = S();                                         // exposition
only
                make-unsigned-like-t(iter_difference_t<I>) size_ = 0;  // exposition
only; present only
                // when StoreSize is true
                public:
                subrange() = default;

                constexpr subrange(I i, S s) requires (!StoreSize);

                constexpr subrange(I i, S s, make-unsigned-like-t(iter_difference_t<I>) n)
                requires (K == subrange_kind::sized);

                template<not-same-as<subrange> R>
                requires forwarding-range<R> &&
                convertible_to<iterator_t<R>, I> && convertible_to<sentinel_t<R>, S>
                constexpr subrange(R&& r) requires (!StoreSize || sized_range<R>);

                template<forwarding-range R>
                requires convertible_to<iterator_t<R>, I> && convertible_to<sentinel_t<R>, S>
                constexpr subrange(R&& r, make-unsigned-like-t(iter_difference_t<I>) n)
                requires (K == subrange_kind::sized)
                : subrange{ranges::begin(r), ranges::end(r), n}
                {}
```

```
        template<not-same-as<subrange> PairLike>
        requires pair-like-convertible-to<PairLike, I, S>
        constexpr subrange(PairLike&& r) requires (!StoreSize)
        : subrange{std::get<0>(std::forward<PairLike>(r)),
                std::get<1>(std::forward<PairLike>(r))}
        {}

        template<pair-like-convertible-to<I, S> PairLike>
        constexpr subrange(PairLike&& r, make-unsigned-like-t(iter_difference_t<I>) n)
        requires (K == subrange_kind::sized)
        : subrange{std::get<0>(std::forward<PairLike>(r)),
                std::get<1>(std::forward<PairLike>(r)), n}
        {}

        template<not-same-as<subrange> PairLike>
        requires pair-like-convertible-from<PairLike, const I&, const S&>
        constexpr operator PairLike() const;

        constexpr I begin() const requires copyable<I>;
        [[nodiscard]] constexpr I begin() requires (!copyable<I>);
        constexpr S end() const;

        constexpr bool empty() const;
        constexpr make-unsigned-like-t(iter_difference_t<I>) size() const
        requires (K == subrange_kind::sized);

        [[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const
        & requires forward_iterator<I>;
        [[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) &&;

        [[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
        requires bidirectional_iterator<I>;
        constexpr subrange& advance(iter_difference_t<I> n);

        friend constexpr I begin(subrange&& r) { return r.begin(); }
        friend constexpr S end(subrange&& r) { return r.end(); }
};

template<input_or_output_iterator I, sentinel_for<I> S>
subrange(I, S, make-unsigned-like-t(iter_difference_t<I>)) ->
subrange<I, S, subrange_kind::sized>;

template<iterator-sentinel-pair P>
subrange(P) -> subrange<tuple_element_t<0, P>, tuple_element_t<1, P>>;

template<iterator-sentinel-pair P>
subrange(P, make-unsigned-like-t(iter_difference_t<tuple_element_t<0, P>>)) ->
subrange<tuple_element_t<0, P>, tuple_element_t<1, P>, subrange_kind::sized>;

template<forwarding-range R>
subrange(R&&) ->
```

```
        subrange<iterator_t<R>, sentinel_t<R>,
        (sized_range<R> || sized_sentinel_for<sentinel_t<R>, iterator_t<R>>)
        ? subrange_kind::sized : subrange_kind::unsized>;

        template<forwarding-range R>
        subrange(R&&, make-unsigned-like-t(range_difference_t<R>)) ->
        subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

        template<size_t N, class I, class S, subrange_kind K>
        requires (N < 2)
        constexpr auto get(const subrange<I, S, K>& r);

        template<size_t N, class I, class S, subrange_kind K>
        requires (N < 2)
        constexpr auto get(subrange<I, S, K>&& r);
}

namespace std {
        using ranges::get;
}
```

Constructors and conversions [range.subrange.ctor]

```
constexpr subrange(I i, S s) requires (!StoreSize);
```

Expects: `[i, s)` is a valid range.

Effects: Initializes `begin_` with ~~`i`~~ `std::move(i)` and `end_` with `s`.

```
constexpr subrange(I i, S s, make-unsigned-like-t(iter_difference_t<I>) n)
requires (K == subrange_kind::sized);
```

Expects: `[i, s)` is a valid range, and `n == make-unsigned-like(ranges::distance(i, s))`.

Effects: Initializes `begin_` with ~~`i`~~ `std::move(i)` and `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `n`.

[*Note:* Accepting the length of the range and storing it to later return from `size()` enables `subrange` to model `sized_range` even when it stores an iterator and sentinel that do not model `sized_sentinel_for`. — *end note*]

```
template<exposition only not-same-as<subrange> R>
requires exposition only forwarding-range<R> &&
convertible_to<iterator_t<R>, I> && convertible_to<sentinel_t<R>, S>
constexpr subrange(R&& r) requires (!StoreSize || sized_range<R>);
```

Effects: Equivalent to:

- If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), ranges::size(r)}`.
- Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

```
template<exposition only not-same-as<subrange> PairLike>
requires exposition only pair-like-convertible-from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
```

Effects: Equivalent to: `return PairLike(begin_, end_);`

Accessors [range.subrange.access]

```
constexpr I begin() const requires copyable<I>;
```

Effects: Equivalent to: `return begin_;`

```
[[nodiscard]] constexpr I begin() requires (!copyable<I>);
```

Effects: Equivalent to: `return std::move(begin_);`

```
constexpr S end() const;
```

Effects: Equivalent to: `return end_;`

```
constexpr bool empty() const;
```

Effects: Equivalent to: `return begin_ == end_;`

```
constexpr make-unsigned-like-t(iter_difference_t<I>) size() const
requires (K == subrange_kind::sized);
```

Effects:

- If `StoreSize` is `true`, equivalent to: `return size_;`
- Otherwise, equivalent to: `return make-unsigned-like-t(end_ - begin_);`

```
[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const
& requires forward_iterator<I>;
```

Effects: Equivalent to:

```
auto tmp = *this;
tmp.advance(n);
return tmp;
```

[*Note:* If `I` does not model `forward_iterator`, `next` can invalidate `*this`. — *end note*]

```
[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) &&;
```

Effects: Equivalent to:

```
advance(n);
return std::move(*this);
```

```
[[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
requires bidirectional_iterator<I>;
```

Effects: Equivalent to:

```
auto tmp = *this;
tmp.advance(-n);
return tmp;
```

```
constexpr subrange& advance(iter_difference_t<I> n);
```

Effects: Equivalent to:

- If `StoreSize` is `true`,

```
auto d = n - ranges::advance(begin_, n, end_);
if (d >= 0)
size_ -= make-unsigned-like(d);
else
size_ += make-unsigned-like(-d);
return *this;
```

- Otherwise,

```
ranges::advance(begin_, n, end_);
return *this;
```

```
template<size_t N, class I, class S, subrange_kind K>
requires (N < 2)
constexpr auto get(const subrange<I, S, K>& r);

template<size_t N, class I, class S, subrange_kind K>
requires (N < 2)
constexpr auto get(subrange<I, S, K>&& r);
```

Effects: Equivalent to:

```
if constexpr (N == 0)
return r.begin();
else
return r.end();
```

🔲 Range adaptors [range.adaptors]

...

🔲 Filter view [range.filter]

🔲 Overview [range.filter.overview]

`filter_view` presents a `view` of an underlying sequence without the elements that fail to satisfy a predicate.

[*Example:*

```
vector<int> is{ 0, 1, 2, 3, 4, 5, 6 };
filter_view evens{is, [](int i) { return 0 == i % 2; }};
        for (int i : evens)
        cout << i << ' '; // prints:  0 2 4 6
```

— *end example*]

🔲 Class template `filter_view` [range.filter.view]

```
namespace std::ranges {
        template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
        requires view<V> && is_object_v<Pred>
        class filter_view : public view_interface<filter_view<V, Pred>> {
                private:
                V base_ = V();                // exposition only
                semiregular-box<Pred> pred_;  // exposition only

                // ??, class filter_view::iterator
                class iterator;               // exposition only
                // ??, class filter_view::sentinel
                class sentinel;               // exposition only

                public:
                filter_view() = default;
                constexpr filter_view(V base, Pred pred);
                template<input_range R>
                requires viewable_range<R> && constructible_from<V, all_view<R>>
                constexpr filter_view(R&& r, Pred pred);

                constexpr V base() const;

                constexpr iterator begin();
                constexpr auto end() {
                        if constexpr (common_range<V>)
                        return iterator{*this, ranges::end(base_)};
                        else
                        return sentinel{*this};
                }
        };

        template<class R, class Pred>
```

```
        filter_view(R&&, Pred) -> filter_view<all_view<R>, Pred>;
}

constexpr filter_view(V base, Pred pred);
```

Effects: Initializes `base_` with `std::move(base)` and initializes `pred_` with `std::move(pred)`.

```
template<input_range R>
requires viewable_range<R> && constructible_from<V, all_view<R>>
constexpr filter_view(R&& r, Pred pred);
```

Effects: Initializes `base_` with `views::all(std::forward<R>(r))` and initializes `pred_` with `std::move(pred)`.

```
constexpr V base() const;
```

Effects: Equivalent to: `return base_;`

```
constexpr iterator begin();
```

Expects: `pred_.has_value()`.

Returns: `{*this, ranges::find_if(base_, ref(*pred_))}`.

Remarks: In order to provide the amortized constant time complexity required by the `range` concept when `filter_view` models `forward_range`, this function caches the result within the `filter_view` for use on subsequent calls.

🔲 Class `filter_view::iterator` [range.filter.iterator]

```
namespace std::ranges {
        template<class V, class Pred>
        class filter_view<V, Pred>::iterator {
                private:
                iterator_t<V> current_ = iterator_t<V>();   // exposition only
                filter_view* parent_ = nullptr;             // exposition only
                public:
                using iterator_concept  = see below;
                using iterator_category = see below;
                using value_type        = range_value_t<V>;
                using difference_type   = range_difference_t<V>;

                iterator() = default;
                constexpr iterator(filter_view& parent, iterator_t<V> current);

                constexpr iterator_t<V> base() const & requires copyable<iterator_t<V>>;
                constexpr iterator_t<V> base() &&;
                constexpr range_reference_t<V> operator*() const;
                constexpr iterator_t<V> operator->() const
                requires has-arrow<iterator_t<V>> && copyable<iterator_t<V>>;

                constexpr iterator& operator++();
```

```
        constexpr void operator++(int);
        constexpr iterator operator++(int) requires forward_range<V>;

        constexpr iterator& operator--() requires bidirectional_range<V>;
        constexpr iterator operator--(int) requires bidirectional_range<V>;

        friend constexpr bool operator==(const iterator& x, const iterator& y)
        requires equality_comparable<iterator_t<V>>;

        friend constexpr range_rvalue_reference_t<V> iter_move(const iterator& i)
        noexcept(noexcept(ranges::iter_move(i.current_)));
        friend constexpr void iter_swap(const iterator& x, const iterator& y)
        noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
        requires indirectly_swappable<iterator_t<V>>;
    };
}
```

Modification of the element a `filter_view::iterator` denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

`iterator::iterator_concept` is defined as follows:

- If `V` models `bidirectional_range`, then `iterator_concept` denotes `bidirectional_iterator_tag`.
- Otherwise, if `V` models `forward_range`, then `iterator_concept` denotes `forward_iterator_tag`.
- Otherwise, `iterator_concept` denotes `input_iterator_tag`.

`iterator::iterator_category` is defined as follows:

- Let `C` denote the type `iterator_traits<iterator_t<V>>::iterator_category`.
- If `C` models `derived_from<bidirectional_iterator_tag>`, then `iterator_category` denotes `bidirectional_iterator_tag`.
- Otherwise, if `C` models `derived_from<forward_iterator_tag>`, then `iterator_category` denotes `forward_iterator_tag`.
- Otherwise, `iterator_category` denotes ~~`input_iterator_tag`~~ `C`.

```
constexpr iterator(filter_view& parent, iterator_t<V> current);
```

Effects: Initializes `current_` with ~~`current`~~ `std::move(current)` and `parent_` with `addressof(parent)`.

```
constexpr iterator_t<V> base() const & requires copyable<iterator_t<V>>;
```

Effects: Equivalent to: `return current_;`

```
constexpr iterator_t<V> base() &&;
```

Effects: Equivalent to: `return std::move(current_);`

```
constexpr range_reference_t<V> operator*() const;
```

Effects: Equivalent to: `return *current_;`

```
constexpr iterator_t<V> operator->() const
requires has-arrow<iterator_t<V>> && copyable<iterator_t<V>>;
```

Effects: Equivalent to: `return current_;`

```
constexpr iterator& operator++();
```

Effects: Equivalent to:

```
current_ = ranges::find_if(std::move(++current_),
     ranges::end(parent_->base_), ref(*parent_->pred_));
return *this;
```

```
constexpr void operator++(int);
```

Effects: Equivalent to `++*this`.

...

🔲 Class `filter_view::sentinel` [range.filter.sentinel]

...

🔲 Transform view [range.transform]

...

🔲 Class template `transform_view::iterator` [range.transform.iterator]

```
namespace std::ranges {
        template<class V, class F>
        template<bool Const>
        class transform_view<V, F>::iterator {
                private:
                using Parent =                                  // exposition only
                conditional_t<Const, const transform_view, transform_view>;
                using Base   =                                  // exposition only
                conditional_t<Const, const V, V>;
                iterator_t<Base> current_ =                     // exposition only
                iterator_t<Base>();
                Parent* parent_ = nullptr;                      // exposition only
                public:
                using iterator_concept  = see below;
                using iterator_category = see below;
                using value_type        =
                remove_cvref_t<invoke_result_t<F&, range_reference_t<Base>>>;
                using difference_type   = range_difference_t<Base>;
```

```
iterator() = default;
constexpr iterator(Parent& parent, iterator_t<Base> current);
constexpr iterator(iterator<!Const> i)
requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;

constexpr iterator_t<Base> base() const & requires copyable<iterator_t<Base>>;
constexpr iterator_t<Base> base() &&;

constexpr decltype(auto) operator*() const
{ return invoke(*parent_->fun_, *current_); }

constexpr iterator& operator++();
constexpr void operator++(int);
constexpr iterator operator++(int) requires forward_range<Base>;

constexpr iterator& operator--() requires bidirectional_range<Base>;
constexpr iterator operator--(int) requires bidirectional_range<Base>;

constexpr iterator& operator+=(difference_type n)
requires random_access_range<Base>;
constexpr iterator& operator-=(difference_type n)
requires random_access_range<Base>;
constexpr decltype(auto) operator[](difference_type n) const
requires random_access_range<Base>
{ return invoke(*parent_->fun_, current_[n]); }

friend constexpr bool operator==(const iterator& x, const iterator& y)
requires equality_comparable<iterator_t<Base>>;

friend constexpr bool operator<(const iterator& x, const iterator& y)
requires random_access_range<Base>;
friend constexpr bool operator>(const iterator& x, const iterator& y)
requires random_access_range<Base>;
friend constexpr bool operator<=(const iterator& x, const iterator& y)
requires random_access_range<Base>;
friend constexpr bool operator>=(const iterator& x, const iterator& y)
requires random_access_range<Base>;
friend constexpr compare_three_way_result_t<iterator_t<Base>>
operator<=>(const iterator& x, const iterator& y)
requires random_access_range<Base> && three_way_comparable<iterator_t<Base>>;

friend constexpr iterator operator+(iterator i, difference_type n)
requires random_access_range<Base>;
friend constexpr iterator operator+(difference_type n, iterator i)
requires random_access_range<Base>;

friend constexpr iterator operator-(iterator i, difference_type n)
requires random_access_range<Base>;
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
requires random_access_range<Base>;
```

```
                friend constexpr decltype(auto) iter_move(const iterator& i)
                noexcept(noexcept(invoke(*i.parent_->fun_, *i.current_)))
                {
                        if constexpr (is_lvalue_reference_v<decltype(*i)>)
                        return std::move(*i);
                        else
                        return *i;
                }

                friend constexpr void iter_swap(const iterator& x, const iterator& y)
                noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
                requires indirectly_swappable<iterator_t<Base>>;
        };
}
```

`iterator::iterator_concept` is defined as follows:

- If `V` models `random_access_range`, then `iterator_concept` denotes `random_access_iterator_tag`.
- Otherwise, if `V` models `bidirectional_range`, then `iterator_concept` denotes `bidirectional_iterator_tag`.
- Otherwise, if `V` models `forward_range`, then `iterator_concept` denotes `forward_iterator_tag`.
- Otherwise, `iterator_concept` denotes `input_iterator_tag`.

Let `C` denote the type `iterator_traits<iterator_t<Base>>::iterator_category`. If `C` models `derived_from<contiguous_iterator_tag>`, then `iterator_category` denotes `random_access_iterator_tag`; otherwise, `iterator_category` denotes `C`.

```
constexpr iterator(Parent& parent, iterator_t<Base> current);
```

Effects: Initializes `current_` with ~~`current`~~ `std::move(current)` and `parent_` with `addressof(parent)`.

```
constexpr iterator(iterator<!Const> i)
requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;
```

Effects: Initializes `current_` with `std::move(i.current_)` and `parent_` with `i.parent_`.

```
constexpr iterator_t<Base> base() const & requires copyable<iterator_t<Base>>;
```

Effects: Equivalent to: `return current_;`

```
constexpr iterator_t<Base> base() &&;
```

Effects: Equivalent to: `return std::move(current_);`

```
constexpr iterator& operator++();
```

Effects: Equivalent to:

```
        ++current_;
        return *this;
```

...

🔲 Take view [range.take]

...

🔲 Take while view [range.take.while]

...

🔲 Class template `drop_view` [range.drop.view]

```
namespace std::ranges {
        template<view R>
        class drop_view : public view_interface<drop_view<R>> {
                public:
                drop_view() = default;
                constexpr drop_view(R base, range_difference_t<R> count);

                constexpr R base() const;

                constexpr auto begin()
                requires (!(simple-view<R> && random_access_range<R>));
                constexpr auto begin() const
                requires random_access_range<const R>;

                constexpr auto end()
                requires (!simple-view<R>)
                { return ranges::end(base_); }

                constexpr auto end() const
                requires range<const R>
                { return ranges::end(base_); }

                constexpr auto size()
                requires sized_range<R>
                {
                        const auto s = ranges::size(base_);
                        const auto c = static_cast<decltype(s)>(count_);
                        return s < c ? 0 : s - c;
                }

                constexpr auto size() const
                requires sized_range<const R>
                {
                        const auto s = ranges::size(base_);
                        const auto c = static_cast<decltype(s)>(count_);
                        return s < c ? 0 : s - c;
```

```
        }
        private:
        R base_;                                    // exposition only
        range_difference_t<R> count_;               // exposition only
    };

    template<class R>
    drop_view(R&&, range_difference_t<R>) -> drop_view<all_view<R>>;
}
```

```
constexpr drop_view(R base, range_difference_t<R> count);
```

Expects: `count >= 0` is `true`.

Effects: Initializes `base_` with `std::move(base)` and `count_` with `count`.

```
constexpr R base() const;
```

Effects: Equivalent to: `return base_;`

```
constexpr auto begin()
requires (!(simple-view<R> && random_access_range<R>));
constexpr auto begin() const
requires random_access_range<const R>;
```

Returns: `ranges::next(ranges::begin(base_), count_, ranges::end(base_))`.

Remarks: In order to provide the amortized constant-time complexity required by the `range` concept when `drop_view` models `forward_range`, the first overload caches the result within the `drop_view` for use on subsequent calls. [*Note:* Without this, applying a `reverse_view` over a `drop_view` would have quadratic iteration complexity. — *end note*]

🔲 `views::drop` [range.drop.adaptor]

The name `views::drop` denotes a range adaptor object. For some subexpressions `E` and `F`, the expression `views::drop(E, F)` is expression-equivalent to `drop_view{E, F}`.

🔲 Drop while view [range.drop.while]

🔲 Overview [range.drop.while.overview]

Given a unary predicate `pred` and a `view r`, `drop_while_view` produces a `view` of the range `[ranges::find_if_not(r, pred), ranges::end(r))`.

[*Example:*

```
constexpr auto source = "  \t   \t   \t   hello there";
auto is_invisible = [](const auto x) { return x == ' ' || x == '\t'; };
auto skip_ws = drop_while_view{source, is_invisible};
for (auto c : skip_ws) {
```

```
        cout << c;                                  // prints hello there with no
leading space
    }
```

— *end example*]

🔲 Class template `drop_while_view` [range.drop.while.view]

```
namespace std::ranges {
    template<view R, class Pred>
    requires input_range<R> && is_object_v<Pred> &&
    indirect_unary_predicate<const Pred, iterator_t<R>>
    class drop_while_view : public view_interface<drop_while_view<R, Pred>> {
        public:
        drop_while_view() = default;
        constexpr drop_while_view(R base, Pred pred);

        constexpr R base() const;
        constexpr const Pred& pred() const;

        constexpr auto begin();

        constexpr auto end()
        { return ranges::end(base_); }

        private:
        R base_;                                    // exposition only
        semiregular-box<Pred> pred_;                // exposition only
    };

    template<class R, class Pred>
    drop_while_view(R&&, Pred) -> drop_while_view<all_view<R>, Pred>;
}
```

`constexpr drop_while_view(R base, Pred pred);`

Effects: Initializes `base_` with `std::move(base)` and `pred_` with `std::move(pred)`.

`constexpr R base() const;`

Effects: Equivalent to: `return base_;`

`constexpr const Pred& pred() const;`

Effects: Equivalent to: `return *pred_;`

`constexpr auto begin();`

Returns: `ranges::find_if_not(base_, cref(*pred_))`.

Remarks: In order to provide the amortized constant-time complexity required by the `range` concept when `drop_while_view` models `forward_range`, the first call caches the result within

the `drop_while_view` for use on subsequent calls. [*Note:* Without this, applying a `reverse_view` over a `drop_while_view` would have quadratic iteration complexity. — *end note*]

🔲 `views::drop_while` [range.drop.while.adaptor]

The name `views::drop_while` denotes a range adaptor object. For some subexpressions `E` and `F`, the expression `views::drop_while(E, F)` is expression-equivalent to `drop_while_view{E, F}`.

🔲 Join view [range.join]

🔲 Class template `join_view::iterator` [range.join.iterator]

```
namespace std::ranges {
        template<class V>
        template<bool Const>
        struct join_view<V>::iterator {
                private:
                using Parent =                                          // exposition only
                conditional_t<Const, const join_view, join_view>;
                using Base   = conditional_t<Const, const V, V>;        // exposition only

                static constexpr bool ref_is_glvalue =                  // exposition only
                is_reference_v<range_reference_t<Base>>;

                iterator_t<Base> outer_ = iterator_t<Base>();           // exposition only
                iterator_t<range_reference_t<Base>> inner_ =            // exposition only
                iterator_t<range_reference_t<Base>>();
                Parent* parent_ = nullptr;                              // exposition only

                constexpr void satisfy();                               // exposition only
                public:
                using iterator_concept  = see below;
                using iterator_category = see below;
                using value_type        = range_value_t<range_reference_t<Base>>;
                using difference_type   = see below;

                iterator() = default;
                constexpr iterator(Parent& parent, iterator_t<V> outer);
                constexpr iterator(iterator<!Const> i)
                requires Const &&
                convertible_to<iterator_t<V>, iterator_t<Base>> &&
                convertible_to<iterator_t<InnerRng>,
                iterator_t<range_reference_t<Base>>>;

                constexpr decltype(auto) operator*() const { return *inner_; }

                constexpr iterator_t<Base> operator->() const
                requires has-arrow<iterator_t<Base>> && copyable<iterator_t<Base>>;
```

```
                constexpr iterator& operator++();
                constexpr void operator++(int);
                constexpr iterator operator++(int)
                requires ref_is_glvalue && forward_range<Base> &&
                forward_range<range_reference_t<Base>>;

                constexpr iterator& operator--()
                requires ref_is_glvalue && bidirectional_range<Base> &&
                bidirectional_range<range_reference_t<Base>>;

                constexpr iterator operator--(int)
                requires ref_is_glvalue && bidirectional_range<Base> &&
                bidirectional_range<range_reference_t<Base>>;

                friend constexpr bool operator==(const iterator& x, const iterator& y)
                requires ref_is_glvalue && equality_comparable<iterator_t<Base>> &&
                equality_comparable<iterator_t<range_reference_t<Base>>>;

                friend constexpr decltype(auto) iter_move(const iterator& i)
                noexcept(noexcept(ranges::iter_move(i.inner_))) {
                        return ranges::iter_move(i.inner_);
                }

                friend constexpr void iter_swap(const iterator& x, const iterator& y)
                noexcept(noexcept(ranges::iter_swap(x.inner_, y.inner_)));
        };
}
```

`iterator::iterator_concept` is defined as follows:

- If `ref_is_glvalue` is `true` and `Base` and `range_reference_t<Base>` each model `bidirectional_range`, then `iterator_concept` denotes `bidirectional_iterator_tag`.
- Otherwise, if `ref_is_glvalue` is `true` and `Base` and `range_reference_t<Base>` each model `forward_range`, then `iterator_concept` denotes `forward_iterator_tag`.
- Otherwise, `iterator_concept` denotes `input_iterator_tag`.

`iterator::iterator_category` is defined as follows:

- Let *OUTERC* denote `iterator_traits<iterator_t<Base>>::iterator_category`, and let *INNERC* denote `iterator_traits<iterator_t<range_reference_t<Base>>>::iterator_category`.
- If `ref_is_glvalue` is `true` and *OUTERC* and *INNERC* each model `derived_from<bidirectional_iterator_tag>`, `iterator_category` denotes `bidirectional_iterator_tag`.
- Otherwise, if `ref_is_glvalue` is `true` and *OUTERC* and *INNERC* each model `derived_from<forward_iterator_tag>`, `iterator_category` denotes `forward_iterator_tag`.
- Otherwise, if *OUTERC* and *INNERC* each model `derived_from<input_iterator_tag>`, `iterator_category` denotes `input_iterator_tag`.

- Otherwise, `iterator_category` denotes `output_iterator_tag`.

...

```
constexpr iterator(Parent& parent, iterator_t<V> outer)
```

Effects: Initializes `outer_` with `std::move(outer)` and `parent_` with `addressof(parent)`; then calls `satisfy()`.

```
constexpr iterator(iterator<!Const> i)
        requires Const &&
        convertible_to<iterator_t<V>, iterator_t<Base>> &&
        convertible_to<iterator_t<InnerRng>,
        iterator_t<range_reference_t<Base>>>;
```

Effects: Initializes `outer_` with `std::move(i.outer_)`, `inner_` with `std::move(i.inner_)`, and `parent_` with `i.parent_`.

```
constexpr iterator_t<Base> operator->() const
requires has-arrow<iterator_t<Base>> && copyable<iterator_t<Base>>;
```

Effects: Equivalent to `return inner_;`

🚧 Split view [range.split]

🚧 Overview [range.split.overview]

`split_view` takes a `view` and a delimiter, and splits the `view` into subranges on the delimiter. The delimiter can be a single element or a `view` of elements.

[*Example:*

```
string str{"the quick brown fox"};
split_view sentence{str, ' '};
for (auto word : sentence) {
        for (char ch : word)
        cout << ch;
        cout << '*';
}
// The above prints:  the*quick*brown*fox*
```

— *end example*]

🚧 Class template `split_view` [range.split.view]

```
namespace std::ranges {
        template<auto> struct require-constant;       // exposition only

        template<class R>
        concept tiny-range =                          // exposition only
```

```
        sized_range<R> &&
        requires { typename require-constant<remove_reference_t<R>::size()>; } &&
        (remove_reference_t<R>::size() <= 1);

        template<input_range V, forward_range Pattern>
        requires view<V> && view<Pattern> &&
        indirectly_comparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
        (forward_range<V> || tiny-range<Pattern>)
        class split_view : public view_interface<split_view<V, Pattern>> {
                private:
                V base_ = V();                              // exposition only
                Pattern pattern_ = Pattern();               // exposition only
                iterator_t<V> current_ = iterator_t<V>();   // exposition only, present only
if !forward_range<V>
                // ??, class template split_view::outer_iterator
                template<bool> struct outer_iterator;       // exposition only
                // ??, class template split_view::inner_iterator
                template<bool> struct inner_iterator;       // exposition only
                public:
                split_view() = default;
                constexpr split_view(V base, Pattern pattern);

                template<input_range R, forward_range P>
                requires constructible_from<V, all_view<R>> &&
                constructible_from<Pattern, all_view<P>>
                constexpr split_view(R&& r, P&& p);

                template<input_range R>
                requires constructible_from<V, all_view<R>> &&
                constructible_from<Pattern, single_view<range_value_t<R>>>
                constexpr split_view(R&& r, range_value_t<R> e);

                constexpr auto begin() {
                        if constexpr (forward_range<V>)
                        return outer_iterator<simple-view<V>>{*this, ranges::begin(base_)};
                        else {
                                current_ = ranges::begin(base_);
                                return outer_iterator<false>{*this};
                        }
                }

                constexpr auto begin() const requires forward_range<V> && forward_range<const V> {
                        return outer_iterator<true>{*this, ranges::begin(base_)};
                }

                constexpr auto end() requires forward_range<V> && common_range<V> {
                        return outer_iterator<simple-view<V>>{*this, ranges::end(base_)};
                }

                constexpr auto end() const {
                        if constexpr (forward_range<V> && forward_range<const V> && common_range<const
```

```
                        return outer_iterator<true>{*this, ranges::end(base_)};
                        else
                        return default_sentinel;
                }
        };

        template<class R, class P>
        split_view(R&&, P&&) -> split_view<all_view<R>, all_view<P>>;

        template<input_range R>
        split_view(R&&, range_value_t<R>)
        -> split_view<all_view<R>, single_view<range_value_t<R>>>;
}

constexpr split_view(V base, Pattern pattern);
```

Effects: Initializes `base_` with `std::move(base)`, and `pattern_` with `std::move(pattern)`.

```
template<input_range R, forward_range P>
requires constructible_from<V, all_view<R>> &&
constructible_from<Pattern, all_view<P>>
constexpr split_view(R&& r, P&& p);
```

Effects: Initializes `base_` with `views::all(std::forward<R>(r))`, and `pattern_` with `views::all(std::forward<P>(p))`.

```
template<input_range R>
requires constructible_from<V, all_view<R>> &&
constructible_from<Pattern, single_view<range_value_t<R>>>
constexpr split_view(R&& r, range_value_t<R> e);
```

Effects: Initializes `base_` with `views::all(std::forward<R>(r))`, and `pattern_` with `single_view{std::move(e)}`.

🔲 Class template `split_view::outer_iterator` [range.split.outer]

```
namespace std::ranges {
        template<class V, class Pattern>
        template<bool Const>
        struct split_view<V, Pattern>::outer_iterator {
                private:
                using Parent =                          // exposition only
                conditional_t<Const, const split_view, split_view>;
                using Base   =                          // exposition only
                conditional_t<Const, const V, V>;
                Parent* parent_ = nullptr;              // exposition only
                iterator_t<Base> current_ =             // exposition only, present only
if V models forward_range
                iterator_t<Base>();

                public:
```

```
                using iterator_concept  =
                conditional_t<forward_range<Base>, forward_iterator_tag, input_iterator_tag>;
                using iterator_category = input_iterator_tag;
                // ??, class split_view::outer_iterator::value_type
                struct value_type;
                using difference_type   = range_difference_t<Base>;

                outer_iterator() = default;
                constexpr explicit outer_iterator(Parent& parent)
                requires (!forward_range<Base>);
                constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
                requires forward_range<Base>;
                constexpr outer_iterator(outer_iterator<!Const> i)
                requires Const && convertible_to<iterator_t<V>, iterator_t<const V>>;

                constexpr value_type operator*() const;

                constexpr outer_iterator& operator++();
                constexpr decltype(auto) operator++(int) {
                        if constexpr (forward_range<Base>) {
                                auto tmp = *this;
                                ++*this;
                                return tmp;
                        } else
                        ++*this;
                }

                friend constexpr bool operator==(const outer_iterator& x, const outer_iterator& y)
                requires forward_range<Base>;

                friend constexpr bool operator==(const outer_iterator& x, default_sentinel_t);
        };
}
```

Many of the following specifications refer to the notional member *current* of `outer_iterator`. *current* is equivalent to `current_` if `V` models `forward_range`, and `parent_->current_` otherwise.

```
constexpr explicit outer_iterator(Parent& parent)
requires (!forward_range<Base>);
```

Effects: Initializes `parent_` with `addressof(parent)`.

```
constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
requires forward_range<Base>;
```

Effects: Initializes `parent_` with `addressof(parent)` and `current_` with ~~current~~ `std::move(current)`.

```
constexpr outer_iterator(outer_iterator<!Const> i)
requires Const && convertible_to<iterator_t<V>, iterator_t<const V>>;
```

Effects: Initializes `parent_` with `i.parent_` and `current_` with `std::move(i.current_)`.

```
constexpr value_type operator*() const;
```

Effects: Equivalent to: `return value_type{*this};`

```
constexpr outer_iterator& operator++();
```

Effects: Equivalent to:

```
const auto end = ranges::end(parent_->base_);
if (current == end) return *this;
const auto [pbegin, pend] = subrange{parent_->pattern_};
if (pbegin == pend) ++current;
else {
        do {
                const auto [b, p] =
                ranges::mismatch(current std::move(current), end, pbegin, pend);
                current = std::move(b);
                if (p == pend) {
                        current = b;  // The pattern matched; skip it
                        break;
                }
        } while (++current != end);
}
return *this;
```

```
friend constexpr bool operator==(const outer_iterator& x, const outer_iterator& y)
requires forward_range<Base>;
```

Effects: Equivalent to: `return x.current_ == y.current_;`

```
friend constexpr bool operator==(const outer_iterator& x, default_sentinel_t);
```

Effects: Equivalent to: `return x.current == ranges::end(x.parent_->base_);`

🔲 Class `split_view::outer_iterator::value_type` [range.split.outer.value]

```
namespace std::ranges {
        template<class V, class Pattern>
        template<bool Const>
        struct split_view<V, Pattern>::outer_iterator<Const>::value_type {
                private:
                outer_iterator i_ = outer_iterator();              // exposition only
                public:
                value_type() = default;
                constexpr explicit value_type(outer_iterator i);

                constexpr inner_iterator<Const> begin() const requires copyable<outer_iterator>;
                constexpr inner_iterator<Const> begin() requires (!copyable<outer_iterator>);
                constexpr default_sentinel_t end() const;
        };
}

constexpr explicit value_type(outer_iterator i);
```

Effects: Initializes `i_` with ~~`i`~~ `std::move(i)`.

```
constexpr inner_iterator<Const> begin() const requires copyable<outer_iterator>;
```

Effects: Equivalent to: `return inner_iterator<Const>{i_};`

```
constexpr inner_iterator<Const> begin() requires (!copyable<outer_iterator>);
```

Effects: Equivalent to: `return inner_iterator<Const>{std::move(i_)};`

```
constexpr default_sentinel_t end() const;
```

Effects: Equivalent to: `return default_sentinel;`

🔲 Class template `split_view::inner_iterator` [range.split.inner]

```
namespace std::ranges {
        template<class V, class Pattern>
        template<bool Const>
        struct split_view<V, Pattern>::inner_iterator {
                private:
                using Base =
                conditional_t<Const, const V, V>;                 // exposition only
                outer_iterator<Const> i_ = outer_iterator<Const>(); // exposition only
                bool incremented_ = false;                          // exposition only
                public:
                using iterator_concept  = typename outer_iterator<Const>::iterator_concept;
                using iterator_category = see below;
                using value_type        = range_value_t<Base>;
                using difference_type   = range_difference_t<Base>;

                inner_iterator() = default;
                constexpr explicit inner_iterator(outer_iterator<Const> i);

                constexpr decltype(auto) operator*() const { return *i_.current; }

                constexpr inner_iterator& operator++();
                constexpr decltype(auto) operator++(int) {
                        if constexpr (forward_range<V>) {
                                auto tmp = *this;
                                ++*this;
                                return tmp;
                        } else
                        ++*this;
                }

                friend constexpr bool operator==(const inner_iterator& x, const inner_iterator& y)
                requires forward_range<Base>;

                friend constexpr bool operator==(const inner_iterator& x, default_sentinel_t);

                friend constexpr decltype(auto) iter_move(const inner_iterator& i)
```

```
                noexcept(noexcept(ranges::iter_move(i.i_.current))) {
                        return ranges::iter_move(i.i_.current);
                }

                friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
                noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
                requires indirectly_swappable<iterator_t<Base>>;
        };
}
```

The *typedef-name* `iterator_category` denotes: `forward_iterator_tag` if `iterator_traits<iterator_t<Base>>::iterator_category` models `derived_from<forward_iterator_tag>`, and `input_iterator_tag` otherwise.

- `forward_iterator_tag` if `iterator_traits<iterator_t<Base>>::iterator_category` models `derived_from<forward_iterator_tag>`;
- otherwise, `iterator_traits<iterator_t<Base>>::iterator_category`.

```
constexpr explicit inner_iterator(outer_iterator<Const> i);
```

Effects: Initializes `i_` with ~~`i`~~ `std::move(i)`.

```
constexpr inner_iterator& operator++() const;
```

Effects: Equivalent to:

```
incremented_ = true;
if constexpr (!forward_range<Base>) {
        if constexpr (Pattern::size() == 0) {
                return *this;
        }
}
++i_.current;
return *this;
```

```
friend constexpr bool operator==(const inner_iterator& x, const inner_iterator& y)
requires forward_range<Base>;
```

Effects: Equivalent to: `return x.i_.current_ == y.i_.current_;`

```
friend constexpr bool operator==(const inner_iterator& x, default_sentinel_t);
```

Effects: Equivalent to:

```
auto cur = x.i_.current;
auto end = ranges::end(x.i_.parent_->base_);
if (cur == end) return true;
auto [pcur, pend] = subrange{x.i_.parent_->pattern_};
if (pcur == pend) return x.incremented_;
do {
        if (*cur != *pcur) return false;
        if (++pcur == pend) return true;
} while (++cur != end);
```

```
        return false;

        auto [pcur, pend] = subrange{x.i_.parent_->pattern_};
        auto end = ranges::end(x.i_.parent_->base_);
        if constexpr (tiny-range<Pattern>) {
                const auto & cur = x.i_.current;
                if (cur == end) return true;
                if (pcur == pend) return x.incremented_;
                return *cur == *pcur;
        }
        else {
                auto cur = x.i_.current;
                if (cur == end) return true;
                if (pcur == pend) return x.incremented_;
                do {
                        if (*cur != *pcur) return false;
                        if (++pcur == pend) return true;
                } while (++cur != end);
                return false;
        }
```

```
friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
requires indirectly_swappable<iterator_t<Base>>;
```

Effects: Equivalent to `ranges::iter_swap(x.i_.current, y.i_.current)`.

🔲 `views::split` [range.split.adaptor]

The name `views::split` denotes a range adaptor object. For some subexpressions `E` and `F`, the expression `views::split(E, F)` is expression-equivalent to `split_view{E, F}`.

🔲 Counted view [range.counted]

A counted view presents a `view` of the elements of the counted range `[i, n)` for some iterator `i` and non-negative integer `n`.

The name `views::counted` denotes a customization point object. Let `E` and `F` be expressions, and let `T` be `decay_t<decltype((E))>`. Then the expression `views::counted(E, F)` is expression-equivalent to:

- If `T` models `input_or_output_iterator` and `decltype((F))` models `convertible_to<iter_difference_t<T>>`,
  - `subrange{E, E + static_cast<iter_difference_t<T>>(F)}` if `T` models `random_access_iterator`.
  - Otherwise, `subrange{counted_iterator{E, F}, default_sentinel}`.

- Otherwise, `views::counted(E, F)` is ill-formed. [*Note:* This case can result in substitution failure when `views::counted(E, F)` appears in the immediate context of a template instantiation. — *end note*]

24.7.? Common view [range.common]

24.7.?.1 Overview [range.common.overview]

`common_view` takes a `view` which has different types for its iterator and sentinel and turns it into a `view` of the same elements with an iterator and sentinel of the same type.

[*Note:* `common_view` is useful for calling legacy algorithms that expect a range's iterator and sentinel types to be the same. — *end note*]

[*Example:*

```
// Legacy algorithm:
template<class ForwardIterator>
size_t count(ForwardIterator first, ForwardIterator last);

template<forward_range R>
void my_algo(R&& r) {
        auto&& common = common_view{r};
        auto cnt = count(common.begin(), common.end());
        // ...
}
```

— *end example*]

24.7.?.2 Class template `common_view` [range.common.view]

```
namespace std::ranges {
template<view V>
requires (!common_range<V> && copyable<iterator_t<V>>)
class common_view : public view_interface<common_view<V>> {
        private:
        V base_ = V();  // exposition only
        public:
        common_view() = default;

        constexpr explicit common_view(V r);

        template<viewable_range R>
        requires (!common_range<R> && constructible_from<V, all_view<R>>)
        constexpr explicit common_view(R&& r);

        constexpr V base() const;

        constexpr auto size() requires sized_range<V> {
```

```
                return ranges::size(base_);
        }
        constexpr auto size() const requires sized_range<const V> {
                return ranges::size(base_);
        }

        constexpr auto begin() {
                if constexpr (random_access_range<V> && sized_range<V>)
                return ranges::begin(base_);
                else
                return common_iterator<iterator_t<V>, sentinel_t<V>>(ranges::begin(base_));
        }

        constexpr auto begin() const requires range<const V> {
                if constexpr (random_access_range<const V> && sized_range<const V>)
                return ranges::begin(base_);
                else
                return common_iterator<iterator_t<const V>, sentinel_t<const V>>(ranges::begin(base_));
        }

        constexpr auto end() {
                if constexpr (random_access_range<V> && sized_range<V>)
                return ranges::begin(base_) + ranges::size(base_);
                else
                return common_iterator<iterator_t<V>, sentinel_t<V>>(ranges::end(base_));
        }

        constexpr auto end() const requires range<const V> {
                if constexpr (random_access_range<const V> && sized_range<const V>)
                return ranges::begin(base_) + ranges::size(base_);
                else
                return common_iterator<iterator_t<const V>, sentinel_t<const V>>(ranges::end(base_));
        }
};

template<class R>
common_view(R&&) -> common_view<all_view<R>>;
}

constexpr explicit common_view(V base);
```

Effects: Initializes `base_` with `std::move(base)`.

```
template<viewable_range R>
requires (!common_range<R> && constructible_from<V, all_view<R>>)
constexpr explicit common_view(R&& r);
```

Effects: Initializes `base_` with `views::all(std::forward<R>(r))`.

```
constexpr V base() const;
```

Effects: Equivalent to: `return base_;`

...

`views::common` [range.common.adaptor]

The name `views::common` denotes a range adaptor object. For some subexpression `E`, the expression `views::common(E)` is expression-equivalent to:

- `views::all(E)`, if `decltype((E))` models `common_range` and `views::all(E)` is a well-formed expression.
- Otherwise, `common_view{E}`.

Reverse view [range.reverse]

...

Istream view [range.istream]

...

Elements view [range.elements]

Overview [range.elements.overview]

`elements_view` takes a `view` of tuple-like values and a `size_t`, and produces a `view` with a value-type of the N^{th} element of the adapted `view`'s value-type.

The name `views::elements<N>` denotes a range adaptor object. For some subexpression `E` and constant expression `N`, the expression `views::elements<N>(E)` is expression-equivalent to `elements_view<all_view<decltype((E))>, N>{E}`.

[*Example:*

```
auto historical_figures = map{
        {"Lovelace"sv, 1815},
        {"Turing"sv, 1912},
        {"Babbage"sv, 1791},
        {"Hamilton"sv, 1936}
};

auto names = historical_figures | views::elements<0>;
for (auto&& name : names) {
        cout << name << ' ';          // prints Babbage Hamilton Lovelace Turing
}

auto birth_years = historical_figures | views::elements<1>;
for (auto&& born : birth_years) {
```

```
        cout << born << ' ';          // prints 1791 1936 1815 1912
}
```

— *end example*]

`keys_view` is an alias for `elements_view<all_view<R>, 0>`, and is useful for extracting keys from associative containers.

[*Example:*

```
auto names = keys_view{historical_figures};
for (auto&& name : names) {
        cout << name << ' ';          // prints Babbage Hamilton Lovelace Turing
}
```

— *end example*]

`values_view` is an alias for `elements_view<all_view<R>, 1>`, and is useful for extracting values from associative containers.

[*Example:*

```
auto is_even = [](const auto x) { return x % 2 == 0; };
        cout << ranges::count_if(values_view{historical_figures}, is_even);     //
prints 2
```

— *end example*]

🔲 Class template `elements_view` [range.elements.view]

```
namespace std::ranges {
        template<class T, size_t N>
        concept has-tuple-element =                    // exposition only
        requires(T t) {
                typename tuple_size<T>::type;
                requires N < tuple_size_v<T>;
                typename tuple_element_t<N, T>;
                { get<N>(t) } -> const tuple_element_t<N, T>&;
        };


        template<input_range R, size_t N>
        requires view<R> && has-tuple-element<range_value_t<R>, N> &&
        has-tuple-element<remove_reference_t<range_reference_t<R>>, N>
        class elements_view : public view_interface<elements_view<R, N>> {
                public:
                elements_view() = default;
                constexpr explicit elements_view(R base);

                constexpr R base() const;
```

```
        constexpr auto begin() requires (!simple-view<R>)
        { return iterator<false>(ranges::begin(base_)); }

        constexpr auto begin() const requires simple-view<R>
        { return iterator<true>(ranges::begin(base_)); }

        constexpr auto end() requires (!simple-view<R>)
        { return ranges::end(base_); }

        constexpr auto end() const requires simple-view<R>
        { return ranges::end(base_); }

        constexpr auto size() requires sized_range<R>
        { return ranges::size(base_); }

        constexpr auto size() const requires sized_range<const R>
        { return ranges::size(base_); }

        private:
        template<bool> struct iterator;                    // exposition only
        R base_ = R();                                     // exposition only
    };
}
```

```
    constexpr explicit elements_view(R base);
```

Effects: Initializes `base_` with `std::move(base)`.

```
    constexpr R base() const;
```

Effects: Equivalent to: `return base_;`

🔲 Class template `elements_view::iterator` [range.elements.iterator]

```
namespace std::ranges {
    template<class R, size_t N>
    template<bool Const>
    class elements_view<R, N>::iterator {                  // exposition only
        using base_t = conditional_t<Const, const R, R>;
        friend iterator<!Const>;

        iterator_t<base_t> current_;
        public:
        using iterator_category = typename iterator_traits<iterator_t<base_t>>::iterator_categor
        using value_type = remove_cvref_t<tuple_element_t<N, range_value_t<base_t>>>;
        using difference_type = range_difference_t<base_t>;

        iterator() = default;
        constexpr explicit iterator(iterator_t<base_t> current);
        constexpr iterator(iterator<!Const> i)
        requires Const && convertible_to<iterator_t<R>, iterator_t<base_t>>;
```

```
constexpr iterator_t<base_t> base() const & requires copyable<iterator_t<base_t>>;
constexpr iterator_t<base_t> base() &&;

constexpr decltype(auto) operator*() const
{ return get<N>(*current_); }

constexpr iterator& operator++();
constexpr void operator++(int) requires (!forward_range<base_t>);
constexpr iterator operator++(int) requires forward_range<base_t>;

constexpr iterator& operator--() requires bidirectional_range<base_t>;
constexpr iterator operator--(int) requires bidirectional_range<base_t>;

constexpr iterator& operator+=(difference_type x)
requires random_access_range<base_t>;
constexpr iterator& operator-=(difference_type x)
requires random_access_range<base_t>;

constexpr decltype(auto) operator[](difference_type n) const
requires random_access_range<base_t>
{ return get<N>(*(current_ + n)); }

friend constexpr bool operator==(const iterator& x, const iterator& y)
requires equality_comparable<iterator_t<base_t>>;
friend constexpr bool operator==(const iterator& x, const sentinel_t<base_t>& y);

friend constexpr bool operator<(const iterator& x, const iterator& y)
requires random_access_range<base_t>;
friend constexpr bool operator>(const iterator& x, const iterator& y)
requires random_access_range<base_t>;
friend constexpr bool operator<=(const iterator& y, const iterator& y)
requires random_access_range<base_t>;
friend constexpr bool operator>=(const iterator& x, const iterator& y)
requires random_access_range<base_t>;
friend constexpr compare_three_way_result_t<iterator_t<base_t>>
operator<=>(const iterator& x, const iterator& y)
requires random_access_range<base_t> && three_way_comparable<iterator_t<base_t>>;

friend constexpr iterator operator+(const iterator& x, difference_type y)
requires random_access_range<base_t>;
friend constexpr iterator operator+(difference_type x, const iterator& y)
requires random_access_range<base_t>;
friend constexpr iterator operator-(const iterator& x, difference_type y)
requires random_access_range<base_t>;
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
requires random_access_range<base_t>;

friend constexpr difference_type
operator-(const iterator<Const>& x, const sentinel_t<base_t>& y)
```

```
            requires sized_sentinel_for<sentinel_t<base_t>, iterator_t<base_t>>;
            friend constexpr difference_type
            operator-(const sentinel_t<base_t>& x, const iterator<Const>& y)
            requires sized_sentinel_for<sentinel_t<base_t>, iterator_t<base_t>>;
    };
}
```

```
    constexpr explicit iterator(iterator_t<base_t> current);
```

Effects: Initializes `current_` with ~~`current`~~ `std::move(current)`.

```
    constexpr iterator(iterator<!Const> i)
    requires Const && convertible_to<iterator_t<R>, iterator_t<base_t>>;
```

Effects: Initializes `current_` with `std::move(i.current_)`.

```
    constexpr iterator_t<base_t> base() const & requires copyable<iterator_t<V>>;
```

Effects: Equivalent to: `return current_;`

```
    constexpr iterator_t<base_t> base() &&;
```

Effects: Equivalent to: `return std::move(current_);`

```
    constexpr iterator& operator++();
```

Effects: Equivalent to:

```
        ++current_;
        return *this;
```

...

References

[P0896] Eric Niebler, Casey Carter, Christopher Di Bella. *The One Ranges Proposal*
https://wg21.link/P0896

[P1456] Casey Carter *Move-only views*
https://wg21.link/P1456

Move-only views (by Casey Carter) (2019-01-25)

[P1035] Christopher Di Bella *Input range adaptors*
https://wg21.link/P1035

[N4820] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/n4820