Document Number: P0789
Date: 2017-10-16
Reply to: Eric Niebler
eric.niebler@gmail.com

Range Adaptors and Utilities

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad fomatting.

Contents

1 General [intro]

"Adopt your own view and adapt with others' views."

—*Mohammed Sekouty*

1.1 Scope [intro.scope]

[Editor's note: For motivation and design considerations, please refer to N4128, "Ranges for the Standard Library, Revision 1" ([2]).]

1 This document provides extensions to the Ranges TS [1] to support the creation of pipelines of range transformations. In particular, changes and extensions to the Ranges TS include:

(1.1) — An `iterator_range` type that stores an iterator/sentinel pair and satisfies the requirements of the `View` concept.

(1.2) — A `sized_iterator_range` type that stores an iterator/sentinel pair and a size, and satisfies the requirements of both the `View` and `SizedRange` concepts.

(1.3) — A `view::all` range adaptor that turns a `Range` into a `View` while respecting memory safety.

(1.4) — A `view::filter` range adaptor that accepts a `Range` and a `Predicate` and returns a `View` of the underlying range that skips those elements that fail to satisfy the predicate.

(1.5) — A `view::transform` range adaptor that accepts a `Range` and a unary `Invocable` and produces a view that applies the invocable to each element of the underlying range.

(1.6) — A `view::iota` range that takes a `WeaklyIncrementable` and yields a range of elements produced by incrementing the initial element monotonically. Optionally, it takes an upper bound at which to stop.

(1.7) — A `view::emtpy` range that creates an empty range of a certain element type.

(1.8) — A `view::single` range that creates a range of cardinality 1 with the specified element.

(1.9) — A `view::join` range adaptor takes a range of ranges, and lazily flattens the ranges into one range.

(1.10) — A `view::split` range adaptor takes a range and a delimiter, and lazily splits the range into subranges on the delimiter. The delimiter may be either an element or a subrange.

1.2 References [intro.refs]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2017, *Programming Languages - C++*

(1.2) — JTC1/SC22/WG21 N4685, *Technical Specification - C++ Extensions for Ranges*

ISO/IEC 14882:2017 is herein called the *C++ Standard* and N4685 is called the *Ranges TS*.

1.3 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in 1.3 in the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.4 Namespaces, headers, and modifications to standard classes [intro.namespaces]

1 Since the extensions described in this document are experimental additions to the Ranges TS, everything defined herein is declared within namespace `std::experimental::ranges::v1`.

2 Unless otherwise specified, references to other entities described in this document are assumed to be qualified with `std::experimental::ranges::`, and references to entities described in the International Standard are assumed to be qualified with `std::`.

10 Ranges library [ranges]

[Editor's note: To the section "Header `<experimental/ranges/range>` synopsis" 10.3 [range.synopsis], add the following:]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 10.7.1:
  template <class D>
  class view_interface;

  // 10.7.2.1:
  template <Iterator I, Sentinel<I> S = I>
  class iterator_range;

  // 10.7.2.2:
  template <Iterator I, Sentinel<I> S = I>
  class sized_iterator_range;

  // 10.8.3:
  namespace view { inline constexpr unspecified  all = unspecified ; }

  template <Range R>
    requires is_lvalue_reference_v<R> || View<decay_t<R>>
  using all_view = decay_t<decltype(view::all(declval<R>()))>;

  // 10.8.4:
  template <InputRange R, IndirectUnaryPredicate<iterator_t<R>> Pred>
    requires View<R>
  class filter_view;

  namespace view { inline constexpr unspecified  filter = unspecified ; }

  // 10.8.6:
  template <InputRange R, CopyConstructible F>
    requires View<R> && Invocable<F&, reference_t<iterator_t<R>>>
  class transform_view;

  namespace view { inline constexpr unspecified  transform = unspecified ; }

  // 10.8.8:
  template <WeaklyIncrementable I, Semiregular Bound = unreachable>
    requires WeaklyEqualityComparable<I, Bound>
  class iota_view;

  namespace view { inline constexpr unspecified  iota = unspecified ; }

  // 10.8.12:
  template <InputRange R>
    requires View<R> && InputRange<reference_t<iterator_t<R>>> &&
      (is_reference_v<reference_t<iterator_t<R>>> ||
       View<value_type_t<iterator_t<R>>>)
  class join_view;
```

```
  namespace view { inline constexpr unspecified  join = unspecified ; }

  // 10.8.14:
  template <class T>
    requires requires { *(T*)nullptr; }
  class empty_view;

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

  // 10.8.15:
  template <CopyConstructible T>
  class single_view;

  namespace view { inline constexpr unspecified  single = unspecified ; }

  // exposition only
  template <class R>
  concept bool tiny-range = see below;

  // 10.8.17:
  template <InputRange Rng, ForwardRange Pattern>
    requires View<Rng> && View<Pattern> &&
        IndirectlyComparable<iterator_t<Rng>, iterator_t<Pattern>> &&
        (ForwardRange<Rng> || tiny-range<Pattern>)
  struct split_view;

  namespace view { inline constexpr unspecified  split = unspecified ; }

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

namespace std {
  template <class I, class S>
    struct tuple_size<ranges::iterator_range<I, S>>
      : std::integral_constant<size_t, 2> {};
  template <class I, class S>
    struct tuple_element<0, ranges::iterator_range<I, S>> {
      using type = I;
    };
  template <class I, class S>
    struct tuple_element<1, ranges::iterator_range<I, S>> {
      using type = S;
    };

  template <class I, class S>
    struct tuple_size<ranges::sized_iterator_range<I, S>>
      : std::integral_constant<size_t, 3> {};
  template <class I, class S>
    struct tuple_element<0, ranges::sized_iterator_range<I, S>> {
      using type = I;
```

```
    };
  template <class I, class S>
    struct tuple_element<1, ranges::sized_iterator_range<I, S>> {
      using type = S;
    };
  template <class I, class S>
    struct tuple_element<2, ranges::sized_iterator_range<I, S>> {
      using type = ranges::difference_type_t<I>;
    };
}
```

[Editor's note: After Ranges TS subclause 10.6 [ranges.requirements], insert a new subclause 10.7, "Range utilities" with stable name [ranges.utilities]]

10.7 Range utilities [ranges.utilities]

1 The components in this section are general utilities for representing and manipulating ranges.

10.7.1 View interface [ranges.view_interface]

1 The `view_interface` is a helper for defining `View`-like types that offer a container-like interface. It is parameterized with the type that inherits from it.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1
{
  // exposition only
  template <Range R>
  struct range-common-iterator-impl {
    using type = common_iterator<iterator_t<R>, sentinel_t<R>>;
  };
  template <BoundedRange R>
  struct range-common-iterator-impl<R> {
    using type = iterator_t<R>;
  };
  template <Range R>
    using range-common-iterator =
      typename range-common-iterator-impl<R>::type;

  template <class D>
  class view_interface : view_base {
  private:
    constexpr D& derived() noexcept { // exposition only
      return static_cast<D&>(*this);
    }
    constexpr const D& derived() const noexcept { // exposition only
      return static_cast<const D&>(*this);
    }
  public:
    constexpr bool empty() const requires ForwardRange<const D>;
    constexpr explicit operator bool() const requires ForwardRange<const D>;
    constexpr bool operator!() const requires ForwardRange<const D>;

    constexpr auto size() const requires ForwardRange<const D> &&
      SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;

    constexpr decltype(auto) front() requires ForwardRange<D>;
    constexpr decltype(auto) front() const requires ForwardRange<const D>;
```

```
    constexpr decltype(auto) back()
      requires BidirectionalRange<D> && BoundedRange<D>;
    constexpr decltype(auto) back() const
      requires BidirectionalRange<const D> && BoundedRange<const D>;

    template <RandomAccessRange R = D>
      constexpr decltype(auto) operator[](difference_type_t<iterator_t<R>> n);
    template <RandomAccessRange R = const D>
      constexpr decltype(auto) operator[](difference_type_t<iterator_t<R>> n) const;

    template <RandomAccessRange R = D>
        requires SizedRange<R>
      constexpr decltype(auto) at(difference_type_t<iterator_t<R>> n);
    template <RandomAccessRange R = const D>
        requires SizedRange<R>
      constexpr decltype(auto) at(difference_type_t<iterator_t<R>> n) const;

    template <ForwardRange C>
        requires !View<C> && MoveConstructible<C> &&
          ConvertibleTo<value_type_t<iterator_t<const D>>, value_type_t<iterator_t<C>>> &&
          Constructible<C, range-common-iterator<const D>, range-common-iterator<const D>>
      operator C () const;
  };
}}}}
```

2 The template parameter for `view_interface` may be an incomplete type.

10.7.1.1 `view_interface` accessors [ranges.view_interface.accessors]

```
constexpr bool empty() const requires ForwardRange<const D>;
```

1 *Effects:* Equivalent to `return ranges::begin(derived()) == ranges::end(derived());`.

```
constexpr explicit operator bool() const requires ForwardRange<const D>;
```

2 *Returns:* `!empty()`.

```
constexpr bool operator!() const requires ForwardRange<const D>;
```

3 *Returns:* `empty()`.

```
constexpr auto size() const requires ForwardRange<const D> &&
  SizedSentinel<sentinel_t<const D>, iterator_t<const D>>;
```

4 *Effects:* Equivalent to `return ranges::end(derived()) - ranges::begin(derived());`.

```
constexpr decltype(auto) front() requires ForwardRange<D>;
constexpr decltype(auto) front() const requires ForwardRange<const D>;
```

5 *Requires:* `!empty()`.

6 *Effects:* Equivalent to `return *ranges::begin(derived());`.

```
constexpr decltype(auto) back()
  requires BidirectionalRange<D> && BoundedRange<D>;
constexpr decltype(auto) back() const
  requires BidirectionalRange<const D> && BoundedRange<const D>;
```

7 *Requires:* `!empty()`.

8 *Effects:* Equivalent to `return *prev(ranges::end(derived()));`.

```
template <RandomAccessRange R = D>
  constexpr decltype(auto) operator[](difference_type_t<iterator_t<R>> n);
template <RandomAccessRange R = const D>
  constexpr decltype(auto) operator[](difference_type_t<iterator_t<R>> n) const;
```

9 *Requires:* `ranges::begin(derived()) + n` is well-formed.

10 *Effects:* Equivalent to `return ranges::begin(derived())[n];`.

```
template <RandomAccessRange R = D>
    requires SizedRange<R>
  constexpr decltype(auto) at(difference_type_t<iterator_t<R>> n);
template <RandomAccessRange R = const D>
    requires SizedRange<R>
  constexpr decltype(auto) at(difference_type_t<iterator_t<R>> n) const;
```

11 *Effects:* Equivalent to `return ranges::begin(derived())[n];`.

12 *Throws:* `out_of_range` if `n < 0 || n >= ranges::size(derived())`.

```
template <ForwardRange C>
    requires !View<C> && MoveConstructible<C> &&
      ConvertibleTo<value_type_t<iterator_t<const D>>, value_type_t<iterator_t<C>>> &&
      Constructible<C, range-common-iterator<const D>, range-common-iterator<const D>>
  operator C () const;
```

13 *Effects:* Equivalent to:

```
using I = range-common-iterator<R>;
return C{I{ranges::begin(derived())}, I{ranges::end(derived())}};
```

10.7.2 Iterator ranges [ranges.iterator.ranges]

1 The `iterator_range` and `sized_iterator_range` classes bundle together an iterator and a sentinel into a single object that satisfies the `View` concept. `sized_iterator_range` additionally stores the range's size and satisfies the `SizedRange` concept.

10.7.2.1 `iterator_range` [ranges.iterator_range]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <Iterator I, Sentinel<I> S = I>
  class iterator_range
    : public tagged_pair<tag::begin(I), tag::end(S)>,
      public view_interface<iterator_range<I, S>> {
  public:
    using iterator = I;
    using sentinel = S;

    iterator_range() = default;
    constexpr iterator_range(I i, S s);

    template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
      constexpr iterator_range(iterator_range<X, Y> r);

    template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
```

```
    constexpr iterator_range(pair<X, Y> r);

  template <Range R>
      requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
    constexpr iterator_range(R& r);

  template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
    constexpr iterator_range& operator=(iterator_range<X, Y> r);

  template <Range R>
      requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
    constexpr iterator_range& operator=(R& r);

  template <Constructible<const I&> X, Constructible<const S&> Y>
      requires ConvertibleTo<const I&, X> && ConvertibleTo<const S&, Y>
    constexpr operator pair<X, Y>() const;

  constexpr bool empty() const;
};

template <Iterator I, Sentinel<I> S>
iterator_range(I, S) -> iterator_range<I, S>;

template <Range R>
iterator_range(R&) -> iterator_range<iterator_t<R>, sentinel_t<R>>;

template <std::size_t N, class I, class S>
    requires N < 2
  constexpr decltype(auto) get(iterator_range<I, S>&& r) noexcept;
template <std::size_t N, class I, class S>
    requires N < 2
  constexpr decltype(auto) get(iterator_range<I, S>& r) noexcept;
template <std::size_t N, class I, class S>
    requires N < 2
  constexpr decltype(auto) get(const iterator_range<I, S>& r) noexcept;
}}}}
```

10.7.2.1.1 `iterator_range` constructors [ranges.iterator_range.ctor]

```
constexpr iterator_range(I i, S s);
```

1 *Effects:* Initializes `tagged_pair<tag::begin(I), tag::end(S)>` with `i` and `s`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr iterator_range(iterator_range<X, Y> r);
```

2 *Effects:* Initializes `tagged_pair<tag::begin(I), tag::end(S)>` with `r.begin()` and `r.end()`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr iterator_range(pair<X, Y> r);
```

3 *Effects:* Initializes `tagged_pair<tag::begin(I), tag::end(S)>` with `r.first` and `r.second`.

```
template <Range R>
    requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
  constexpr iterator_range(R& r);
```

4 *Effects:* Initializes `tagged_pair<tag::begin(I), tag::end(S)>` with `ranges::begin(r)` and `ranges::end(r)`.

10.7.2.1.2 `iterator_range` operators [ranges.iterator_range.ops]

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr iterator_range& operator=(iterator_range<X, Y> r);
```

1 *Effects:* Equivalent to:

```
this->first = r.begin();
this->second = r.end();
return *this;
```

```
template <Range R>
    requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
  constexpr iterator_range& operator=(R& r);
```

2 *Effects:* Equivalent to:

```
this->first = ranges::begin(r);
this->second = ranges::end(r);
return *this;
```

```
template <Constructible<const I&> X, Constructible<const S&> Y>
    requires ConvertibleTo<const I&, X> && ConvertibleTo<const S&, Y>
  constexpr operator pair<X, Y>() const;
```

3 *Effects:* Equivalent to `return {this->first, this->second};`.

10.7.2.1.3 `iterator_range` accessors [ranges.iterator_range.accessors]

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

1 *Effects:* Equivalent to `this->first == this->second`.

10.7.2.1.4 `iterator_range` non-member functions [ranges.iterator_range.nonmember]

```
template <std::size_t N, class I, class S>
  requires N < 2
constexpr decltype(auto) get(iterator_range<I, S>&& r) noexcept;
template <std::size_t N, class I, class S>
  requires N < 2
constexpr decltype(auto) get(iterator_range<I, S>& r) noexcept;
template <std::size_t N, class I, class S>
  requires N < 2
constexpr decltype(auto) get(const iterator_range<I, S>& r) noexcept;
```

1 *Effects:* Equivalent to:

```
if constexpr (N == 0)
  return static_cast<decltype(r)&&>(r).begin();
else
  return static_cast<decltype(r)&&>(r).end();
```

10.7.2.2 `sized_iterator_range` [ranges.sized_iterator_range]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <Iterator I, Sentinel<I> S = I>
  class sized_iterator_range
    : public view_interface<sized_iterator_range<I, S>> {
  private:
```

```
    iterator_range<I, S> rng_; // exposition only
    difference_type_t<I> size_; // exposition only
  public:
    using iterator = I;
    using sentinel = S;

    sized_iterator_range() = default;
    constexpr sized_iterator_range(I i, S s, difference_type_t<I> n);

    template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
      constexpr sized_iterator_range(pair<X, Y> r, difference_type_t<I> n);
    template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
      constexpr sized_iterator_range(iterator_range<X, Y> r, difference_type_t<I> n);
    template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
      constexpr sized_iterator_range(sized_iterator_range<X, Y> r);
    template <SizedRange R>
        requires Constructible<iterator_range<I, S>, R&>
      constexpr sized_iterator_range(R& r);

    template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
      constexpr sized_iterator_range& operator=(sized_iterator_range<X, Y> r);
    template <SizedRange R>
        requires Assignable<iterator_range<I, S>&, R&>
      constexpr sized_iterator_range& operator=(R& r);

    template <Constructible<const I&> X, Constructible<const S&> Y>
        requires ConvertibleTo<const I&, X> && ConvertibleTo<const S&, Y>
      constexpr operator pair<X, Y>() const;
    template <Constructible<const I&> X, Constructible<const S&> Y>
        requires ConvertibleTo<const I&, X> && ConvertibleTo<const S&, Y>
      constexpr operator iterator_range<X, Y>() const;

    constexpr operator iterator_range<I, S> const &() const & noexcept;

    constexpr I begin() const;
    constexpr S end() const;
    constexpr difference_type_t<I> size() const noexcept;
  };

  template <Iterator I, Sentinel<I> S>
  sized_iterator_range(I, S, difference_type_t<I>) -> sized_iterator_range<I, S>;

  template <SizedRange R>
  explicit sized_iterator_range(R&) -> sized_iterator_range<iterator_t<R>, sentinel_t<R>>;

  template <std::size_t N, class I, class S>
      requires N < 3
    constexpr auto get(const sized_iterator_range<I, S>& r);
}}}}
```

10.7.2.2.1 `sized_iterator_range` constructors [ranges.sized_iterator_range.ctor]

```
constexpr sized_iterator_range(I i, S s, difference_type_t<I> n);
```

1 *Requires:* `distance(i, s) == n`.

2 *Effects:* Initializes `rng_` with `i` and `s`, and initializes `size_` with `n`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr sized_iterator_range(pair<X, Y> r, difference_type_t<I> n);
```

3 *Requires:* `distance(r.first, r.second) == n`.

4 *Effects:* Initializes `rng_` with `r.first` and `r.second`, and initializes `size_` with `n`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr sized_iterator_range(iterator_range<X, Y> r, difference_type_t<I> n);
```

5 *Requires:* `distance(r.begin(), r.end()) == n`.

6 *Effects:* Initializes `rng_` with `r.begin()` and `r.end()`, and initializes `size_` with `n`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr sized_iterator_range(sized_iterator_range<X, Y> r);
```

7 *Effects:* Initializes `rng_` with `r.begin()` and `r.end()`, and initializes `size_` with `r.size()`.

```
template <SizedRange R>
    requires Constructible<iterator_range<I, S>, R&>
  constexpr sized_iterator_range(R& r);
```

8 *Effects:* Initializes `rng_` with `r`, and initializes `size_` with `ranges::size(r)`.

10.7.2.2.2 `sized_iterator_range` operators [ranges.sized_iterator_range.ops]

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y>
  constexpr sized_iterator_range& operator=(sized_iterator_range<X, Y> r);
```

1 *Effects:* Equivalent to:

```
rng_.first = r.begin();
rng_.second = r.end();
size_ = r.size();
return *this;
```

```
template <SizedRange R>
    requires Assignable<iterator_range<I, S>&, R&>
  constexpr sized_iterator_range& operator=(R& r);
```

2 *Effects:* Equivalent to:

```
rng_ = r;
size_ = ranges::size(r);
return *this;
```

```
template <Constructible<const I&> X, Constructible<const S&> Y>
    requires ConvertibleTo<const I&, X> && ConvertibleTo<const S&, Y>
  constexpr operator pair<X, Y>() const;
```

3 *Effects:* Equivalent to `return {rng_.first, rng_.second};`.

```
template <Constructible<const I&> X, Constructible<const S&> Y>
    requires ConvertibleTo<const I&, X> && ConvertibleTo<const S&, Y>
  constexpr operator iterator_range<X, Y>() const;
```

4 *Effects:* Equivalent to `return {rng_.first, rng_.second};`.

```
constexpr operator iterator_range<I, S> const &() const & noexcept;
```

5 *Effects:* Equivalent to `return rng_;`.

10.7.2.2.3 `sized_iterator_range` accessors [ranges.sized__iterator__range.accessors]

```
constexpr I begin() const;
```

1 *Effects:* Equivalent to `return rng_.first;`.

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

2 *Effects:* Equivalent to `return rng_.second;`.

```
constexpr difference_type_t<I> size() const noexcept;
```

3 *Effects:* Equivalent to `return size_;`.

10.7.2.2.4 `sized_iterator_range` non-member functions [ranges.iterator__range.nonmember]

```
template <std::size_t N, class I, class S>
    requires N < 3
  constexpr auto get(const sized_iterator_range<I, S>& r);
```

1 *Effects:* Equivalent to:

```
if constexpr (N == 0)
  return r.begin();
else if constexpr (N == 1)
  return r.end();
else
  return r.size();
```

10.8 Range adaptors [ranges.adaptors]

1 This section defines *range adaptors*, which are utilities that transform a `Range` into a `View` with custom behaviors. These adaptors can be chained to create pipelines of range transformations that evaluate lazily as the resulting view is iterated.

2 Range adaptors are declared in namespace `std::experimental::ranges::v1::view`.

3 The bitwise or operator is overloaded for the purpose of creating adaptor chain pipelines. The adaptors also support function call syntax with equivalent semantics.

4 [*Example:*

```
vector<int> ints{0,1,2,3,4,5};
auto even = [](int i){ return 0 == i % 2; };
auto square = [](int i) { return i * i; };
for (int i : ints | view::filter(even) | view::transform(square)) {
  cout << i << ' '; // prints: 0 4 16
}
```

— *end example*]

10.8.1 Range adaptor objects [ranges.adaptor.object]

1 A *range adaptor object* is a customization point object () that accepts a `Range` as its first argument and that returns a `View`. If the adaptor accepts only one argument, then the following alternate syntaxes are semantically equivalent:

```
adaptor(rng)
rng | adaptor
```

If the adaptor accepts more than one argument, then the following alternate syntaxes are semantically equivalent:

```
adaptor(rng, args...)
rng | adaptor(args...)
```

2 The first argument to a range adaptor shall be either an lvalue `Range` or a `View`.

10.8.2 Semiregular wrapper [ranges.adaptor.semiregular_wrapper]

1 Many of the types in this section are specified in terms of an exposition-only helper called *semiregular*`<T>`. This type behaves exactly like `optional<T>` with the following exceptions:

(1.1) — *semiregular*`<T>` constrains its argument with `CopyConstructible<T>`.

(1.2) — If `T` satisfies `DefaultConstructble`, the default constructor of *semiregular*`<T>` is equivalent to:

```
constexpr semiregular()
    noexcept(is_nothrow_default_constructible<T>::value)
  : semiregular{in_place} {}
```

(1.3) — If the syntactic requirements of `Assignable<T&, const T&>` are not satisfied, the copy assignment operator is equivalent to:

```
constexpr semiregular& operator=(const semiregular& that)
    noexcept(is_nothrow_copy_constructible<T>::value) {
  if (that) emplace(*that)
  else reset();
  return *this;
}
```

(1.4) — If the syntactic requirements of `Assignable<T&, T>` are not satisfied, the move assignment operator is equivalent to:

```
constexpr semiregular& operator=(semiregular&& that)
    noexcept(is_nothrow_move_constructible<T>::value) {
  if (that) emplace(std::move(*that))
  else reset();
  return *this;
}
```

10.8.3 `view::all` [ranges.adaptors.all]

1 The purpose of `view::all` is to return a `View` that includes all elements of the `Range` passed in.

2 The name `view::any` denotes a range adaptor object (10.8.1). Given an expression `E` and a type `T` such that `decltype((E))` is `T`, then the expression `view::all(E)` for some subexpression `E` is expression-equivalent to:

(2.1) — *DECAY_COPY*`(E)` if the type of `E` satisfies the concept `View`.

(2.2) — `sized_iterator_range{E}` if `E` is an lvalue and has a type that satisfies concept `SizedRange`.

(2.3) — `iterator_range{E}` if `E` is an lvalue and has a type that satisfies concept `Range`.

(2.4) — Otherwise, `view::all(E)` is ill-formed.

10.8.4 Class template `filter_view` [ranges.adaptors.filter_view]

1 The purpose of `filter_view` is to present a view of an underlying sequence without the elements that fail to satisfy a predicate.

2 [*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*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <InputRange R, IndirectUnaryPredicate<iterator_t<R>> Pred>
    requires View<R>
  class filter_view : view_interface<filter_view<R, Pred>> {
  private:
    R base_; // exposition only
    semiregular<Pred> pred_; // exposition only
  public:
    filter_view() = default;
    constexpr filter_view(R base, Pred pred);
    template <InputRange O>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
        Constructible<R, all_view<O>>
    constexpr filter_view(O&& o, Pred pred);

    constexpr R base() const;

    class iterator;
    class sentinel;

    constexpr iterator begin();
    constexpr sentinel end();
    constexpr iterator end() requires BoundedRange<R>;
  };

  template <InputRange R, CopyConstructible Pred>
    requires IndirectUnaryPredicate<Pred, iterator_t<R>> &&
      (is_lvalue_reference_v<R> || View<decay_t<R>>)
  filter_view(R&&, Pred) -> filter_view<all_view<R>, Pred>;
}}}}
```

10.8.4.1 `filter_view` operations [ranges.adaptors.filter_view.ops]

10.8.4.1.1 `filter_view` constructors [ranges.adaptors.filter_view.ctor]

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

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

```
template <InputRange O>
  requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
    Constructible<R, all_view<O>>
constexpr filter_view(O&& o, Pred pred);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and initializes `pred_` with `std::move(pred)`.

10.8.4.1.2 `filter_view` conversion [ranges.adaptors.filter_view.conv]

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

1 *Returns:* `base_`.

10.8.4.1.3 `filter_view` range begin [ranges.adaptors.filter_view.begin]

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

1 *Effects:* Equivalent to:

```
return {*this, ranges::find_if(base_, ref(*pred_))};
```

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

10.8.4.1.4 `filter_view` range end [ranges.adaptors.filter_view.end]

```
constexpr sentinel end();
```

1 *Returns:* `sentinel{*this}`.

```
constexpr iterator end() requires BoundedRange<R>;
```

2 *Returns:* `iterator{*this, ranges::end(base_)}`.

10.8.4.2 Class template `filter_view::iterator` [ranges.adaptors.filter_view.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R, class Pred>
  class filter_view<R, Pred>::iterator {
  private:
    iterator_t<R> current_ {}; // exposition only
    filter_view* parent_ = nullptr; // exposition only
  public:
    using iterator_category = see below;
    using value_type = value_type_t<iterator_t<R>>;
    using difference_type = difference_type_t<iterator_t<R>>;

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

    constexpr iterator_t<R> base() const;
    constexpr reference_t<iterator_t<R>> operator*() const;

    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int) requires ForwardRange<R>;

    constexpr iterator& operator--() requires BidirectionalRange<R>;
    constexpr iterator operator--(int) requires BidirectionalRange<R>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
      requires EqualityComparable<iterator_t<R>>;
    friend constexpr bool operator!=(const iterator& x, const iterator& y)
      requires EqualityComparable<iterator_t<R>>;

    friend constexpr rvalue_reference_t<iterator_t<R>> iter_move(const iterator& i)
      noexcept(see below);
```

```
    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(see below) requires IndirectlySwappable<iterator_t<R>>;
  };
}}}}
```

1 The type `filter_view<R>::iterator::iterator_category` is defined as follows:

(1.1) — If `R` satisfies `BidirectionalRange<R>`, then `iterator_category` is an alias for `ranges::bidirectional_iterator_tag`.

(1.2) — If `R` satisfies `ForwardRange<R>`, then `iterator_category` is an alias for `ranges::forward_iterator_tag`.

(1.3) — Otherwise, `iterator_category` is an alias for `ranges::input_iterator_tag`.

10.8.4.2.1 `filter_view::iterator` operations [ranges.adaptors.filter_view.iterator]

10.8.4.2.1.1 `filter_view::iterator` constructors [ranges.adaptors.filter_view.iterator.ctor]

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

1 *Effects:* Initializes `current_` with `current` and `parent_` with `&parent`.

10.8.4.2.1.2 `filter_view::iterator` conversion [ranges.adaptors.filter_view.iterator.conv]

```
constexpr iterator_t<R> base() const;
```

1 *Returns:* `current_`.

10.8.4.2.1.3 `filter_view::iterator::operator*` [ranges.adaptors.filter_view.iterator.star]

```
constexpr reference_t<iterator_t<R>> operator*() const;
```

1 *Returns:* `*current_`.

10.8.4.2.1.4 `filter_view::iterator::operator++` [ranges.adaptors.filter_view.iterator.inc]

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

1 *Effects:* Equivalent to:

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

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

2 *Effects:* Equivalent to `(void)++*this`.

```
constexpr iterator operator++(int) requires ForwardRange<R>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

10.8.4.2.1.5 `filter_view::iterator::operator--` [ranges.adaptors.filter_view.iterator.dec]

```
constexpr iterator& operator--() requires BidirectionalRange<R>;
```

1 *Effects:* Equivalent to:

```
do
  --current_;
while(invoke(*parent_->pred_, *current_));
return *this;
```

```
constexpr iterator operator--(int) requires BidirectionalRange<R>;
```

2 *Effects:* Equivalent to:

```
auto tmp = *this;
--*this;
return tmp;
```

10.8.4.2.1.6 `filter_view::iterator` comparisons [ranges.adaptors.filter_view.iterator.comp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<R>>;
```

1 *Returns:* `x.current_ == y.current_`.

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<iterator_t<R>>;
```

2 *Returns:* `!(x == y)`.

10.8.4.2.2 `filter_view::iterator` non-member functions [ranges.adaptors.filter_view.iterator.nonmember]

```
friend constexpr rvalue_reference_t<iterator_t<R>> iter_move(const iterator& i)
  noexcept(see below);
```

1 *Returns:* `ranges::iter_move(i.current_)`.

2 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_move(i.current_))
```

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  noexcept(see below) requires IndirectlySwappable<iterator_t<R>>;
```

3 *Effects:* Equivalent to `ranges::iter_swap(x.current_, y.current_)`.

4 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_swap(x.current_, y.current_))
```

10.8.4.3 Class template `filter_view::sentinel` [ranges.adaptors.filter_view.sentinel]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R, class Pred>
  class filter_view<R, Pred>::sentinel {
  private:
    sentinel_t<R> end_; // exposition only
  public:
    sentinel() = default;
```

```
    explicit constexpr sentinel(filter_view& parent);

    constexpr sentinel_t<R> base() const;

    friend constexpr bool operator==(const iterator& x, const sentinel& y);
    friend constexpr bool operator==(const sentinel& x, const iterator& y);
    friend constexpr bool operator!=(const iterator& x, const sentinel& y);
    friend constexpr bool operator!=(const sentinel& x, const iterator& y);
  };
}}}}
```

10.8.4.3.1 `filter_view::sentinel` constructors [ranges.adaptors.filter_view.sentinel.ctor]

```
explicit constexpr sentinel(filter_view& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent)`.

10.8.4.3.2 `filter_view::sentinel` conversion [ranges.adaptors.filter_view.sentinel.conv]

```
constexpr sentinel_t<R> base() const;
```

1 *Returns:* `end_`.

10.8.4.3.3 `filter_view::sentinel` comparison [ranges.adaptors.filter_view.sentinel.comp]

```
friend constexpr bool operator==(const iterator& x, const sentinel& y);
```

1 *Returns:* `x.current_ == y.end_`.

```
friend constexpr bool operator==(const sentinel& x, const iterator& y);
```

2 *Returns:* `y == x`.

```
friend constexpr bool operator!=(const iterator& x, const sentinel& y);
```

3 *Returns:* `!(x == y)`.

```
friend constexpr bool operator!=(const sentinel& x, const iterator& y);
```

4 *Returns:* `!(y == x)`.

10.8.5 `view::filter` [ranges.adaptors.filter]

1 The name `view::filter` denotes a range adaptor object (10.8.1). Let `E` and `P` be expressions such that types `T` and `U` are `decltype((E))` and `decltype((P))` respectively. Then the expression `view::filter(E, P)` is expression-equivalent to:

(1.1) — `filter_view{E, P}` if `InputRange<T> && IndirectUnaryPredicate<decay_t<U>, iterator_t<T>>` is satisfied.

(1.2) — Otherwise, `view::filter(E, P)` is ill-formed.

10.8.6 Class template `transform_view` [ranges.adaptors.transform_view]

1 The purpose of `transform_view` is to present a view of an underlying sequence after applying a transformation function to each element.

2 [*Example:*

```
vector<int> is{ 0, 1, 2, 3, 4 };
transform_view squares{is, [](int i) { return i * i; }};
for (int i : squares)
  cout << i << ' '; // prints: 0 1 4 9 16
```

— end example]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <InputRange R, CopyConstructible F>
    requires View<R> && Invocable<F&, reference_t<iterator_t<R>>>
  class transform_view : view_interface<transform_view<R, F>> {
  private:
    R base_; // exposition only
    semiregular<F> fun_; // exposition only
    template <bool Const>
      struct __iterator; // exposition only
    template <bool Const>
      struct __sentinel; // exposition only
  public:
    transform_view() = default;
    constexpr transform_view(R base, F fun);
    template <InputRange O>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
          Constructible<R, all_view<O>>
    constexpr transform_view(O&& o, F fun);

    using iterator = __iterator<false>;
    using const_iterator = __iterator<true>;
    using sentinel = __sentinel<false>;
    using const_sentinel = __sentinel<true>;

    constexpr R base() const;

    constexpr iterator begin();
    constexpr const_iterator begin() const requires Range<const R> &&
      Invocable<const F&, reference_t<iterator_t<const R>>>;

    constexpr sentinel end();
    constexpr const_sentinel end() const requires Range<const R> &&
      Invocable<const F&, reference_t<iterator_t<const R>>>;
    constexpr iterator end() requires BoundedRange<R>;
    constexpr const_iterator end() const requires BoundedRange<const R> &&
      Invocable<const F&, reference_t<iterator_t<const R>>>;

    constexpr auto size() requires SizedRange<R>;
    constexpr auto size() const requires SizedRange<const R>;
  };

  template <class R, class F>
  transform_view(R&& r, F fun) -> transform_view<all_view<R>, F>;
}}}}
```

10.8.6.1 `transform_view` operations [ranges.adaptors.transform_view.ops]

10.8.6.1.1 `transform_view` constructors [ranges.adaptors.transform_view.ctor]

```
constexpr transform_view(R base, F fun);
```

1 *Effects:* Initializes `base_` with `std::move(base)` and initializes `fun_` with `std::move(fun)`.

```
template <InputRange O>
  requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
```

```
      Constructible<R, all_view<O>>
constexpr transform_view(O&& o, F fun);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and initializes `fun_` with `std::move(fun)`.

10.8.6.1.2 `transform_view` conversion [ranges.adaptors.transform_view.conv]

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

1 *Returns:* `base_`.

10.8.6.1.3 `transform_view` range begin [ranges.adaptors.transform_view.begin]

```
constexpr iterator begin();
constexpr const_iterator begin() const
  requires Range<const R> && Invocable<const F&, reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to:

```
return {*this, ranges::begin(base_)};
```

10.8.6.1.4 `transform_view` range end [ranges.adaptors.transform_view.end]

```
constexpr sentinel end();
constexpr const_sentinel end() const requires Range<const R> &&
    Invocable<const F&, reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to `sentinel{ranges::end(base_)}` and `const_sentinel{ranges::end(base_)}` for the first and second overload, respectively.

```
constexpr iterator end() requires BoundedRange<R>;
constexpr const_iterator end() const requires BoundedRange<R> &&
    Invocable<const F&, reference_t<iterator_t<const R>>>;
```

2 *Effects:* Equivalent to:

```
return {*this, ranges::end(base_)};
```

10.8.6.1.5 `transform_view` range size [ranges.adaptors.transform_view.size]

```
constexpr auto size() requires SizedRange<R>;
constexpr auto size() const requires SizedRange<const R>;
```

1 *Returns:* `ranges::size(base_)`.

10.8.6.2 Class template `transform_view::__iterator` [ranges.adaptors.transform_view.iterator]

1 `transform_view<R, F>::__iterator` is an exposition-only type.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R, class F>
    template <bool Const>
      class transform_view<R, F>::__iterator { // exposition only
      private:
        using Parent = conditional_t<Const, const transform_view, transform_view>;
        using Base = conditional_t<Const, const R, R>;
        iterator_t<Base> current_ {};
        Parent* parent_ = nullptr;
      public:
        using iterator_category = iterator_category_t<iterator_t<Base>>;
```

```
  using value_type = remove_const_t<remove_reference_t<
      invoke_result_t<F&, reference_t<iterator_t<Base>>>>>;
  using difference_type = difference_type_t<iterator_t<Base>>;

  __iterator() = default;
  constexpr __iterator(Parent& parent, iterator_t<Base> current);
  constexpr __iterator(__iterator<!Const> i)
    requires Const && ConvertibleTo<iterator_t<R>, iterator_t<Base>>;

  constexpr iterator_t<Base> base() const;
  constexpr decltype(auto) operator*() const;

  constexpr __iterator& operator++();
  constexpr void operator++(int);
  constexpr __iterator operator++(int) requires ForwardRange<Base>;

  constexpr __iterator& operator--() requires BidirectionalRange<Base>;
  constexpr __iterator operator--(int) requires BidirectionalRange<Base>;

  constexpr __iterator& operator+=(difference_type n)
    requires RandomAccessRange<Base>;
  constexpr __iterator& operator-=(difference_type n)
    requires RandomAccessRange<Base>;
  constexpr decltype(auto) operator[](difference_type n) const
    requires RandomAccessRange<Base>;

  friend constexpr bool operator==(const __iterator& x, const __iterator& y)
    requires EqualityComparable<iterator_t<Base>>;
  friend constexpr bool operator!=(const __iterator& x, const __iterator& y)
    requires EqualityComparable<iterator_t<Base>>;

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

  friend constexpr __iterator operator+(__iterator i, difference_type n)
    requires RandomAccessRange<Base>;
  friend constexpr __iterator operator+(difference_type n, __iterator i)
    requires RandomAccessRange<Base>;

  friend constexpr __iterator operator-(__iterator i, difference_type n)
    requires RandomAccessRange<Base>;
  friend constexpr difference_type operator-(const __iterator& x, const __iterator& y)
    requires RandomAccessRange<Base>;

  friend constexpr decltype(auto) iter_move(const __iterator& i)
    noexcept(see below);
  friend constexpr void iter_swap(const __iterator& x, const __iterator& y)
    noexcept(see below) requires IndirectlySwappable<iterator_t<Base>>;
};
```

```
}}}}
```

10.8.6.2.1 `transform_view::__iterator` operations [ranges.adaptors.transform_view.iterator.ops]

10.8.6.2.1.1 `transform_view::__iterator` constructors [ranges.adaptors.transform_view.iterator.ctor]

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

1 *Effects:* Initializes `current_` with `current` and initializes `parent_` with `&parent`.

```
constexpr __iterator(__iterator<!Const> i)
  requires Const && ConvertibleTo<iterator_t<R>, iterator_t<Base>>;
```

2 *Effects:* Initializes `parent_` with `i.parent_` and `current_` with `i.current_`.

10.8.6.2.1.2 `transform_view::__iterator` conversion [ranges.adaptors.transform_view.iterator.conv]

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

1 *Returns:* `current_`.

10.8.6.2.1.3 `transform_view::__iterator::operator*` [ranges.adaptors.transform_view.iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Returns:* `invoke(*parent_->fun_, *current_)`.

10.8.6.2.1.4 `transform_view::__iterator::operator++` [ranges.adaptors.transform_view.iterator.inc]

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

1 *Effects:* Equivalent to:

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

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

2 *Effects:* Equivalent to:

```
++current_;
```

```
constexpr __iterator operator++(int) requires ForwardRange<Base>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

10.8.6.2.1.5 `transform_view::__iterator::operator--` [ranges.adaptors.transform_view.iterator.dec]

```
constexpr __iterator& operator--() requires BidirectionalRange<Base>;
```

1 *Effects:* Equivalent to:

```
--current_;
return *this;
```

```
constexpr __iterator operator--(int) requires BidirectionalRange<Base>;
```

2 *Effects:* Equivalent to:

```
auto tmp = *this;
--*this;
return tmp;
```

10.8.6.2.1.6 `transform_view::__iterator` advance [ranges.adaptors.transform_view.iterator.adv]

```
constexpr __iterator& operator+=(difference_type n)
  requires RandomAccessRange<Base>;
```

1 *Effects:* Equivalent to:

```
current_ += n;
return *this;
```

```
constexpr __iterator& operator-=(difference_type n)
  requires RandomAccessRange<Base>;
```

2 *Effects:* Equivalent to:

```
current_ -= n;
return *this;
```

10.8.6.2.1.7 `transform_view::__iterator` index [ranges.adaptors.transform_view.iterator.idx]

```
constexpr decltype(auto) operator[](difference_type n) const
  requires RandomAccessRange<Base>;
```

1 *Effects:* Equivalent to:

```
return invoke(*parent_->fun_, current_[n]);
```

10.8.6.2.2 `transform_view::__iterator` comparisons [ranges.adaptors.transform_view.iterator.comp]

```
friend constexpr bool operator==(const __iterator& x, const __iterator& y)
  requires EqualityComparable<iterator_t<Base>>;
```

1 *Returns:* `x.current_ == y.current_`.

```
friend constexpr bool operator!=(const __iterator& x, const __iterator& y)
  requires EqualityComparable<iterator_t<Base>>;
```

2 *Returns:* `!(x == y)`.

```
friend constexpr bool operator<(const __iterator& x, const __iterator& y)
  requires RandomAccessRange<Base>;
```

3 *Returns:* `x.current_ < y.current_`.

```
friend constexpr bool operator>(const __iterator& x, const __iterator& y)
  requires RandomAccessRange<Base>;
```

4 *Returns:* `y < x`.

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

5 *Returns:* `!(y < x)`.

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

6 *Returns:* `!(x < y)`.

10.8.6.2.3 `transform_view::__iterator` non-member functions [ranges.adaptors.transform_view.iterator.nonmember]

```
friend constexpr __iterator operator+(__iterator i, difference_type n)
  requires RandomAccessRange<Base>;
friend constexpr __iterator operator+(difference_type n, __iterator i)
  requires RandomAccessRange<Base>;
```

1 *Returns:* `__iterator{*i.parent_, i.current_ + n}`.

```
friend constexpr __iterator operator-(__iterator i, difference_type n)
  requires RandomAccessRange<Base>;
```

2 *Returns:* `__iterator{*i.parent_, i.current_ - n}`.

```
friend constexpr difference_type operator-(const __iterator& x, const __iterator& y)
  requires RandomAccessRange<Base>;
```

3 *Returns:* `x.current_ - y.current_`.

```
friend constexpr decltype(auto) iter_move(const __iterator& i)
  noexcept(see below);
```

4 *Effects:* Equivalent to:

(4.1) — If the expression `*i` is an lvalue, then `std::move(*i)`.

(4.2) — Otherwise, `*i`.

5 *Remarks:* The expression in the `noexcept` is equivalent to:

```
noexcept(invoke(*i.parent_->fun_, *i.current_))
```

```
friend constexpr void iter_swap(const __iterator& x, const __iterator& y)
  noexcept(see below) requires IndirectlySwappable<iterator_t<Base>>;
```

6 *Effects:* Equivalent to `ranges::iter_swap(x.current_, y.current_)`.

7 *Remarks:* The expression in the `noexcept` is equivalent to:

```
noexcept(ranges::iter_swap(x.current_, y.current_))
```

10.8.6.3 Class template `transform_view::__sentinel` [ranges.adaptors.transform_view.sentinel]

1 `transform_view<R, F>::__sentinel` is an exposition-only type.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R, class F>
    template <bool Const>
      class transform_view<R, F>::__sentinel {
      private:
        using Parent = conditional_t<Const, const transform_view, transform_view>;
        using Base = conditional_t<Const, const R, R>;
        sentinel_t<Base> end_ {};
      public:
        __sentinel() = default;
        explicit constexpr __sentinel(sentinel_t<Base> end);
        constexpr __sentinel(__sentinel<!Const> i)
          requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;

        constexpr sentinel_t<Base> base() const;

        friend constexpr bool operator==(const __iterator<Const>& x, const __sentinel& y);
        friend constexpr bool operator==(const __sentinel& x, const __iterator<Const>& y);
        friend constexpr bool operator!=(const __iterator<Const>& x, const __sentinel& y);
        friend constexpr bool operator!=(const __sentinel& x, const __iterator<Const>& y);

        friend constexpr difference_type_t<iterator_t<Base>>
          operator-(const __iterator<Const>& x, const __sentinel& y)
            requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
        friend constexpr difference_type_t<iterator_t<Base>>
          operator-(const __sentinel& y, const __iterator<Const>& x)
            requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
      };
}}}}
```

10.8.6.4 `transform_view::__sentinel` constructors [ranges.adaptors.transform_view.sentinel.ctor]

```
explicit constexpr __sentinel(sentinel_t<Base> end);
```

1 *Effects:* Initializes `end_` with `end`.

```
constexpr __sentinel(__sentinel<!Const> i)
  requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes `end_` with `i.end_`.

10.8.6.5 `transform_view::__sentinel` conversion [ranges.adaptors.transform_view.sentinel.conv]

```
constexpr sentinel_t<Base> base() const;
```

1 *Returns:* `end_`.

10.8.6.6 `transform_view::__sentinel` comparison [ranges.adaptors.transform_view.sentinel.comp]

```
friend constexpr bool operator==(const __iterator<Const>& x, const __sentinel& y);
```

1 *Returns:* `x.current_ == y.end_`.

```
friend constexpr bool operator==(const __sentinel& x, const __iterator<Const>& y);
```

2 *Returns:* `y == x`.

```
friend constexpr bool operator!=(const __iterator<Const>& x, const __sentinel& y);
```

3 *Returns:* `!(x == y)`.

```
friend constexpr bool operator!=(const __sentinel& x, const __iterator<Const>& y);
```

4 *Returns:* `!(y == x)`.

10.8.6.7 `transform_view::__sentinel` non-member functions [ranges.adaptors.transform_view.sentinel.nonmember]

```
friend constexpr difference_type_t<iterator_t<Base>>
  operator-(const __iterator<Const>& x, const __sentinel& y)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

1 *Returns:* `x.current_ - y.end_`.

```
friend constexpr difference_type_t<iterator_t<Base>>
  operator-(const __sentinel& y, const __iterator<Const>& x)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

2 *Returns:* `x.end_ - y.current_`.

10.8.7 `view::transform` [ranges.adaptors.transform]

1 The name `view::transform` denotes a range adaptor object (10.8.1). Let `E` and `F` be expressions such that types `T` and `U` are `decltype((E))` and `decltype((F))` respectively. Then the expression `view::transform(E, F)` is expression-equivalent to:

(1.1) — `transform_view{E, F}` if `InputRange<T> && CopyConstructible<decay_t<U>> && Invocable<decay_t<U>&, reference_t<iterator_t<T>>>` is satisfied.

(1.2) — Otherwise, `view::transform(E, F)` is ill-formed.

10.8.8 Class template `iota_view` [ranges.adaptors.iota_view]

1 The purpose of `iota_view` is to generate a sequence of elements by monotonically incrementing an initial value.

[Editor's note: The following definition of `iota_view` presumes the resolution of stl2#507 (https://github.com/ericniebler/stl2/issues/507).]

2 [*Example:*

```
iota_view indices{1, 10};
for (int i : squares)
  cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // exposition only
  template <class I>
  concept bool decrementable = see below;
  // exposition only
```

```
  template <class I>
  concept bool advanceable = see below;

  template <WeaklyIncrementable I, class Bound = unreachable>
    requires WeaklyEqualityComparable<I, Bound>
  struct iota_view : view_interface<iota_view<I, Bound>> {
  private:
    I value_ {}; // exposition only
    I bound_ {}; // exposition only
  public:
    iota_view() = default;
    constexpr explicit iota_view(I value) requires Same<Bound, unreachable>
    constexpr iota_view(I value, Bound bound);

    struct iterator;
    struct sentinel;

    constexpr iterator begin() const;
    constexpr sentinel end() const;
    constexpr iterator end() const requires Same<I, Bound>;

    constexpr auto size() const requires see below;
  };

  template <WeaklyIncrementable I>
  explicit iota_view(I) -> iota_view<I>;

  template <Incrementable I>
  iota_view(I, I) -> iota_view<I, I>;

  template <WeaklyIncrementable I, Semiregular Bound>
  requires WeaklyEqualityComparable<I, Bound> && !ConvertibleTo<Bound, I>
  iota_view(I, Bound) -> iota_view<I, Bound>;
}}}}
```

[3] The exposition-only *decrementable* concept is equivalent to:

```
template <class I>
concept bool decrementable =
  Incrementable<I> && requires(I i) {
    { --i } -> Same<I>&;
    { i-- } -> Same<I>&&;
  };
```

[4] When an object is in the domain of both pre- and post-decrement, the object is said to be *decrementable*.

[5] Let `a` and `b` be incrementable and decrementable objects of type `I`. *decrementable*`<I>` is satisfied only if

(5.1) — `&(--a) == &a`;

(5.2) — If `bool(a == b)` then `bool(a-- == b)`.

(5.3) — If `bool(a == b)` then `bool((a--, a) == --b)`.

(5.4) — If `bool(a == b)` then `bool(-(++a) == b)` and `bool(++(--a) == b)`.

[6] The exposition-only *advanceable* concept is equivalent to:

```
template <class I>
concept bool advanceable =
  decrementable<I> && StrictTotallyOrdered<I> &&
    requires { typename difference_type_t<I>; } &&
    requires(I a, const I b, const difference_type_t<I> n) {
      { a += n } -> Same<I>&;
      { b + n } -> Same<I>&&;
      { n + b } -> Same<I>&&;
      { a -= n } -> Same<I>&;
      { b - n } -> Same<I>&&;
      { b - b } -> Same<difference_type_t<I>>&&;
    };
```

Let `a` and `b` be objects of type `I` such that `b` is reachable from `a`. Let `n` be the smallest number of applications of `++a` necessary to make `bool(a == b)` be `true`. Then if `n` is representable by `difference_type_t<I>`, *`advanceable`*`<I>` is satisfied only if:

(6.1) — `(a += n)` is equal to `b`.

(6.2) — `&(a += n)` is equal to `&a`.

(6.3) — `(a + n)` is equal to `(a += n)`.

(6.4) — For any two positive integers `x` and `y`, if `a + (x + y)` is valid, then `a + (x + y)` is equal to `(a + x) + y`.

(6.5) — `a + 0` is equal to `a`.

(6.6) — If `(a + (n - 1))` is valid, then `a + n` is equal to `++(a + (n - 1))`.

(6.7) — `(b += -n)` is equal to `a`.

(6.8) — `(b -= n)` is equal to `a`.

(6.9) — `&(b -= n)` is equal to `&b`.

(6.10) — `(b - n)` is equal to `(b -= n)`.

(6.11) — `b - a` is equal to `n`.

(6.12) — `a - b` is equal to `-n`.

(6.13) — `a <= b`.

10.8.8.1 `iota_view` operations [ranges.adaptors.iota_view.ops]

10.8.8.1.1 `iota_view` constructors [ranges.adaptors.iota_view.ctor]

```
constexpr explicit iota_view(I value) requires Same<Bound, unreachable>
```

1 *Effects:* Initializes `value_` with `value`.

```
constexpr iota_view(I value, Bound bound);
```

2 *Requires:* `bound` is reachable from `value`.

3 *Effects:* Initializes `value_` with `value` and `bound_` with `bound`.

10.8.8.1.2 `iota_view` range begin [ranges.adaptors.iota_view.begin]

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

1 *Returns:* `iterator{value_}`.

10.8.8.1.3 `iota_view` range end [ranges.adaptors.iota_view.end]

```
constexpr sentinel end() const;
```

1 *Returns:* `sentinel{bound_}`.

```
constexpr iterator end() const requires Same<I, Bound>;
```

2 *Returns:* `iterator{bound_}`.

10.8.8.1.4 `iota_view` range end [ranges.adaptors.iota_view.end]

```
constexpr auto size() const requires see below;
```

1 *Returns:* `bound_ - value_`.

2 *Remarks:* The expression in the `requires` clause is equivalent to:

```
(Same<I, Bound> && advanceable<I>) ||
(Integral<I> && Integral<Bound>) ||
SizedSentinel<Bound, I>
```

10.8.8.2 Class `iota_view::iterator` [ranges.adaptors.iota_view.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class I, class Bound>
  struct iota_view<I, Bound>::iterator {
  private:
    I value_ {}; // exposition only
  public:
    using iterator_category = see below;
    using value_type = I;
    using difference_type = difference_type_t<I>;

    iterator() = default;
    explicit constexpr iterator(I value);

    constexpr I operator*() const noexcept(is_nothrow_copy_constructible_v<I>);

    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int) requires Incrementable<I>;

    constexpr iterator& operator--() requires decrementable<I>;
    constexpr iterator operator--(int) requires decrementable<I>;

    constexpr iterator& operator+=(difference_type n)
      requires advanceable<I>;
    constexpr iterator& operator-=(difference_type n)
      requires advanceable<I>;
    constexpr I operator[](difference_type n) const
      requires advanceable<I>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
```

```
      requires EqualityComparable<I>;
    friend constexpr bool operator!=(const iterator& x, const iterator& y)
      requires EqualityComparable<I>;

    friend constexpr bool operator<(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;
    friend constexpr bool operator>(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;
    friend constexpr bool operator<=(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;
    friend constexpr bool operator>=(const iterator& x, const iterator& y)
      requires StrictTotallyOrdered<I>;

    friend constexpr iterator operator+(iterator i, difference_type n)
      requires advanceable<I>;
    friend constexpr iterator operator+(difference_type n, iterator i)
      requires advanceable<I>;

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

1 `iota_view<I, Bound>::iterator::iterator_category` is defined as follows:

(1.1) — If `I` satisfies *`advanceable`*, then `iterator_category` is `ranges::random_access_iterator_tag`.

(1.2) — Otherwise, if `I` satisfies *`decrementable`*, then `iterator_category` is `ranges::bidirectional_iterator_tag`.

(1.3) — Otherwise, if `I` satisfies `Incrementable`, then `iterator_category` is `ranges::forward_iterator_tag`.

(1.4) — Otherwise, `iterator_category` is `ranges::input_iterator_tag`.

2 [*Note:* Overloads for `iter_move` and `iter_swap` are omitted intentionally. — *end note*]

10.8.8.2.1 `iota_view::iterator` operations [ranges.adaptors.iota_view.iterator.ops]

10.8.8.2.1.1 `iota_view::iterator` constructors [ranges.adaptors.iota_view.iterator.ctor]

```
explicit constexpr iterator(I value);
```

1 *Effects:* Initializes `value_` with `value`.

10.8.8.2.1.2 `iota_view::iterator::operator*` [ranges.adaptors.iota_view.iterator.star]

```
constexpr I operator*() const noexcept(is_nothrow_copy_constructible_v<I>);
```

1 *Returns:* `value_`.

2 [*Note:* The `noexcept` clause is needed by the default `iter_move` implementation. — *end note*]

10.8.8.2.1.3 `iota_view::iterator::operator++` [ranges.adaptors.iota_view.iterator.inc]

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

1 *Effects:* Equivalent to:

```
++value_;
return *this;
```

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

2 *Effects:* Equivalent to `++*this`.

```
constexpr iterator operator++(int) requires Incrementable<I>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

10.8.8.2.1.4 `iota_view::iterator::operator--` [ranges.adaptors.iota_view.iterator.dec]

```
constexpr iterator& operator--() requires decrementable<I>;
```

1 *Effects:* Equivalent to:

```
--value_;
return *this;
```

```
constexpr iterator operator--(int) requires decrementable<I>;
```

2 *Effects:* Equivalent to:

```
auto tmp = *this;
--*this;
return tmp;
```

10.8.8.2.1.5 `iota_view::iterator` advance [ranges.adaptors.iota_view.iterator.adv]

```
constexpr iterator& operator+=(difference_type n)
  requires advanceable<I>;
```

1 *Effects:* Equivalent to:

```
value_ += n;
return *this;
```

```
constexpr iterator& operator-=(difference_type n)
  requires advanceable<I>;
```

2 *Effects:* Equivalent to:

```
value_ -= n;
return *this;
```

10.8.8.2.1.6 `iota_view::iterator` index [ranges.adaptors.iota_view.iterator.idx]

```
constexpr I operator[](difference_type n) const
  requires advanceable<I>;
```

1 *Returns:* `value_ + n`.

10.8.8.2.2 `iota_view::iterator` comparisons [ranges.adaptors.iota_view.iterator.cmp]

```
friend constexpr bool operator==(const iterator& x, const iterator& y)
  requires EqualityComparable<I>;
```

1 *Returns:* `x.value_ == y.value_`.

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires EqualityComparable<I>;
```

2 *Returns:* `!(x == y)`.

```
friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

3 *Returns:* `x.value_ < y.value_`.

```
friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

4 *Returns:* `y < x`.

```
friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

5 *Returns:* `!(y < x)`.

```
friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires StrictTotallyOrdered<I>;
```

6 *Returns:* `!(x < y)`.

10.8.8.2.3 `iota_view::iterator` non-member functions [ranges.adaptors.iota_view.iterator.nonmember]

```
friend constexpr iterator operator+(iterator i, difference_type n)
  requires advanceable<I>;
```

1 *Returns:* `iterator{*i + n}`.

```
friend constexpr iterator operator+(difference_type n, iterator i)
  requires advanceable<I>;
```

2 *Returns:* `i + n`.

```
friend constexpr iterator operator-(iterator i, difference_type n)
  requires advanceable<I>;
```

3 *Returns:* `i + -n`.

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires advanceable<I>;
```

4 *Returns:* `*x - *y`.

10.8.8.3 Class `iota_view::sentinel` [ranges.adaptors.iota_view.sentinel]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class I, class Bound>
  struct iota_view<I, Bound>::sentinel {
  private:
    Bound bound_ {}; // exposition only
  public:
    sentinel() = default;
    constexpr explicit sentinel(Bound bound);

    friend constexpr bool operator==(const iterator& x, const sentinel& y);
    friend constexpr bool operator==(const sentinel& x, const iterator& y);
    friend constexpr bool operator!=(const iterator& x, const sentinel& y);
    friend constexpr bool operator!=(const sentinel& x, const iterator& y);
  };
}}}}
```

10.8.8.3.1 `iota_view::sentinel` constructors [ranges.adaptors.iota_view.ctor]

```
constexpr explicit sentinel(Bound bound);
```

1 *Effects:* Initializes `bound_` with `bound`.

10.8.8.3.2 `iota_view::sentinel` comparisons [ranges.adaptors.iota_view.comp]

```
friend constexpr bool operator==(const iterator& x, const sentinel& y);
```

1 *Returns:* `x.value_ == y.bound_`.

```
friend constexpr bool operator==(const sentinel& x, const iterator& y);
```

2 *Returns:* `y == x`.

```
friend constexpr bool operator!=(const iterator& x, const sentinel& y);
```

3 *Returns:* `!(x == y)`.

```
friend constexpr bool operator!=(const sentinel& x, const iterator& y);
```

4 *Returns:* `!(y == x)`.

10.8.9 `view::iota` [ranges.adaptors.iota]

1 The name `view::iota` denotes a customization point object (). Let `E` and `F` be expressions such that their un-*cv* qualified types are `I` and `J` respectively. Then the expression `view::iota(E)` is expression-equivalent to:

(1.1) — `iota_view{E}` if `WeaklyIncrementable<I>` is satisfied.

(1.2) — Otherwise, `view::iota(E)` is ill-formed.

2 The expression `view::iota(E, F)` is expression-equivalent to:

(2.1) — `iota_view{E, F}` if either of the following sets of constraints is satisfied:

(2.1.1) — `Incrementable<I> && Same<I, J>`

(2.1.2) — `WeaklyIncrementable<I> && Semiregular<J> && WeaklyEqualityComparable<I, J> && !ConvertibleTo<J, I>`

(2.2) — Otherwise, `view::iota(E, F)` is ill-formed.

10.8.10 Class template `take_view` [ranges.adaptors.take_view]

1 The purpose of `take_view` is to produce a range of the first *N* elements from another range.

2 [*Example:*

```
vector<int> is{0,1,2,3,4,5,6,7,8,9};
take_view few{is, 5};
for (int i : few)
  cout << i << ' '; // prints: 0 1 2 3 4
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <InputRange R>
    requires View<R>
  struct take_view : view_interface<take_view<R>> {
  private:
    R base_ {}; // exposition only
    difference_type_t<iterator_t<R>> count_ {}; // exposition only
    template <bool Const>
      struct __sentinel; // exposition only
  public:
    take_view() = default;
    constexpr take_view(R base, difference_type_t<iterator_t<R>> count);
    template <InputRange O>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
          Constructible<R, all_view<O>>
    constexpr take_view(O&& o, difference_type_t<iterator_t<R>> count);

    constexpr R base() const;

    constexpr auto begin();
    constexpr auto begin() const requires Range<const R>;
    constexpr auto begin() requires RandomAccessRange<R> && SizedRange<R>;
    constexpr auto begin() const
      requires RandomAccessRange<const R> && SizedRange<const R>;

    constexpr auto end();
    constexpr auto end() const requires Range<const R>;
    constexpr auto end() requires RandomAccessRange<R> && SizedRange<R>;
    constexpr auto end() const
      requires RandomAccessRange<const R> && SizedRange<const R>;

    constexpr auto size() requires SizedRange<R>;
    constexpr auto size() const requires SizedRange<const R>;

    using iterator = iterator_t<take_view>;
    using const_iterator = see below;
    using sentinel = sentinel_t<take_view>;
    using const_sentinel = see below;
  };

  template <InputRange R>
  take_view(R&& base, difference_type_t<iterator_t<R>> n)
    -> take_view<all_view<R>>;
}}}}
```

3 `take_view<R>::const_iterator` is defined as follows:

(3.1) — If `const R` satisfies `Range` then `const_iterator` is an alias for `iterator_t<const take_view>`.

(3.2) — Otherwise, there is no type `take_view<R>::const_iterator`.

4 `take_view<R>::const_sentinel` is defined as follows:

(4.1) — If `const R` satisfies `Range` then `const_sentinel` is an alias for `sentinel_t<const take_view>`.

(4.2) — Otherwise, there is no type `take_view<R>::const_sentinel`.

10.8.10.1 `take_view` operations [ranges.adaptors.take_view.ops]

10.8.10.1.1 `take_view` constructors [ranges.adaptors.take_view.ctor]

```
constexpr take_view(R base, difference_type_t<iterator_t<R>> count);
```

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

```
template <InputRange O>
  requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
      Constructible<R, all_view<O>>
constexpr take_view(O&& o, difference_type_t<iterator_t<R>> count);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))` and initializes `count_` with `count`.

10.8.10.1.2 `take_view` conversion [ranges.adaptors.take_view.conv]

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

1 *Returns:* `base_`.

10.8.10.1.3 `take_view` range begin [ranges.adaptors.take_view.begin]

```
constexpr auto begin();
constexpr auto begin() const requires Range<const R>;
```

1 *Effects:* Equivalent to:

```
return make_counted_iterator(ranges::begin(base_), count_);
```

```
constexpr auto begin() requires RandomAccessRange<R> && SizedRange<R>;
constexpr auto begin() const
  requires RandomAccessRange<const R> && SizedRange<const R>;
```

2 *Effects:* Equivalent to:

```
return ranges::begin(base_);
```

10.8.10.1.4 `take_view` range end [ranges.adaptors.take_view.end]

```
constexpr auto end();
constexpr auto end() const requires Range<const R>;
```

1 *Effects:* Equivalent to `__sentinel<false>{ranges::end(base_)}` and `__sentinel<true>{ranges::end(base_-)}` for the first and second overload, respectively.

```
constexpr auto end() requires RandomAccessRange<R> && SizedRange<R>;
constexpr auto end() const
  requires RandomAccessRange<const R> && SizedRange<const R>;
```

2 *Effects:* Equivalent to:

```
return ranges::begin(base_) + size();
```

10.8.10.1.5 `take_view` range size [ranges.adaptors.take_view.size]

```
constexpr auto size() requires SizedRange<R>;
constexpr auto size() const requires SizedRange<const R>;
```

1 *Effects:* Equivalent to `ranges::size(base_) < count_ ? ranges::size(base_) : count_`, except with only one call to `ranges::size(base_)`.

10.8.10.2 Class template `take_view::__sentinel` [ranges.adaptors.take_view.sentinel]

1 `take_view<R>::__sentinel` is an exposition-only type.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R>
    template <bool Const>
      class take_view<R>::__sentinel { // exposition only
      private:
        using Parent = conditional_t<Const, const take_view, take_view>;
        using Base = conditional_t<Const, const R, R>;
        sentinel_t<Base> end_ {};
        using CI = counted_iterator<iterator_t<Base>>;
      public:
        __sentinel() = default;
        constexpr explicit __sentinel(sentinel_t<Base> end);
        constexpr __sentinel(__sentinel<!Const> s)
          requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;

        constexpr sentinel_t<Base> base() const;

        friend constexpr bool operator==(const __sentinel& x, const CI& y)
          requires EqualityComparable<iterator_t<Base>>;
        friend constexpr bool operator==(const CI& x, const __sentinel& y)
          requires EqualityComparable<iterator_t<Base>>;
        friend constexpr bool operator!=(const __sentinel& x, const CI& y)
          requires EqualityComparable<iterator_t<Base>>;
        friend constexpr bool operator!=(const CI& x, const __sentinel& y)
          requires EqualityComparable<iterator_t<Base>>;
      };
}}}}
```

10.8.10.2.1 `take_view::__sentinel` operations [ranges.adaptors.take_view.sentinel.ops]

10.8.10.2.1.1 `take_view::__sentinel` constructors [ranges.adaptors.take_view.sentinel.ctor]

```
constexpr explicit __sentinel(sentinel_t<Base> end);
```

1 *Effects:* Initializes `end_` with `end`.

```
constexpr __sentinel(__sentinel<!Const> s)
  requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes `end_` with `s.end_`.

10.8.10.2.1.2 `take_view::__sentinel` conversion [ranges.adaptors.take_view.sentinel.conv]

```
constexpr sentinel_t<Base> base() const;
```

1 *Returns:* `end_`.

10.8.10.2.2 `take_view::__sentinel` comparisons [ranges.adaptors.take_view.sentinel.comp]

```
friend constexpr bool operator==(const __sentinel& x, const CI& y)
  requires EqualityComparable<iterator_t<Base>>;
```

1 *Returns:* `y.count() == 0 || y.base() == x.end_`.

```
friend constexpr bool operator==(const CI& x, const __sentinel& y)
  requires EqualityComparable<iterator_t<Base>>;
```

2 *Returns:* `y == x`.

```
friend constexpr bool operator!=(const __sentinel& x, const CI& y)
  requires EqualityComparable<iterator_t<Base>>;
```

3 *Returns:* `!(x == y)`.

```
friend constexpr bool operator!=(const CI& x, const __sentinel& y)
  requires EqualityComparable<iterator_t<Base>>;
```

4 *Returns:* `!(y == x)`.

10.8.11 `view::take` [ranges.adaptors.take]

1 The name `view::take` denotes a range adaptor object (10.8.1). Let `E` and `F` be expressions such that type `T` is `decltype((E))`. Then the expression `view::take(E, F)` is expression-equivalent to:

(1.1) — `take_view{E, F}` if `InputRange<T>` is satisfied and if `F` is implicitly convertible to `difference_type_t<iterator_t<T>>`.

(1.2) — Otherwise, `view::take(E, F)` is ill-formed.

10.8.12 Class template `join_view` [ranges.adaptors.join_view]

1 The purpose of `join_view` is to flatten a range of ranges into a range.

2 [*Example:*

```
vector<string> ss{"hello", " ", "world", "!"};
join_view greeting{ss};
for (char ch : greeting)
  cout << ch; // prints: hello world!
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <InputRange R>
      requires View<R> && InputRange<reference_t<iterator_t<R>>> &&
          (is_reference_v<reference_t<iterator_t<R>>> ||
          View<value_type_t<iterator_t<R>>>)
  class join_view : view_interface<join_view<R>> {
  private:
    using InnerRng = reference_t<iterator_t<R>>; // exposition only
    template <bool Const>
      struct __iterator; // exposition only
    template <bool Const>
```

```
      struct __sentinel; // exposition only

    R base_ {}; // exposition only
    all_view<InnerRng> inner_ {}; // exposition only, only present when !is_reference_v<InnerRng>
  public:
    join_view() = default;
    constexpr explicit join_view(R base);

    template <InputRange O>
        requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
            Constructible<R, all_view<O>>
      constexpr explicit join_view(O&& o);

    using iterator = __iterator<false>;
    using const_iterator = __iterator<true>;
    using sentinel = __sentinel<false>;
    using const_sentinel = __sentinel<true>;

    constexpr iterator begin();

    constexpr const_iterator begin() const requires InputRange<const R> &&
        is_reference_v<reference_t<iterator_t<const R>>>;

    constexpr sentinel end();

    constexpr const_sentinel end() const requires InputRange<const R> &&
        is_reference_v<reference_t<iterator_t<const R>>>;

    constexpr iterator end() requires ForwardRange<R> &&
        is_reference_v<InnerRng> && ForwardRange<InnerRng> &&
        BoundedRange<R> && BoundedRange<InnerRng>;

    constexpr const_iterator end() const requires ForwardRange<const R> &&
        is_reference_v<reference_t<iterator_t<const R>>> &&
        ForwardRange<reference_t<iterator_t<const R>>> &&
        BoundedRange<const R> && BoundedRange<reference_t<iterator_t<const R>>>;
  };

  template <InputRange R>
      requires InputRange<reference_t<iterator_t<R>>> &&
          (is_reference_v<reference_t<iterator_t<R>>> ||
          View<value_type_t<iterator_t<R>>>)
    explicit join_view(R&&) -> join_view<all_view<R>>;
}}}}
```

10.8.12.1 `join_view` operations [ranges.adaptors.join_view.ops]

10.8.12.1.1 `join_view` constructors [ranges.adaptors.join_view.ctor]

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

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

```
template <InputRange O>
    requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
        Constructible<R, all_view<O>>
  constexpr explicit join_view(O&& o);
```

2 *Effects:* Initializes `base_` with `view::all(std::forward<O>(o))`.

10.8.12.1.2 `join_view` range begin [ranges.adaptors.join_view.begin]

```
constexpr iterator begin();
constexpr const_iterator begin() const requires InputRange<const R> &&
    is_reference_v<reference_t<iterator_t<const R>>>;
```

1 *Returns:* `{*this, ranges::begin(base_)}`.

10.8.12.1.3 `join_view` range end [ranges.adaptors.join_view.end]

```
constexpr sentinel end();
constexpr const_sentinel end() const requires InputRange<const R> &&
    is_reference_v<reference_t<iterator_t<const R>>>;
```

1 *Effects:* Equivalent to `sentinel{*this}` and `const_sentinel{*this}` for the first and second overload, respectively.

```
constexpr iterator end() requires ForwardRange<R> &&
    is_reference_v<InnerRng> && ForwardRange<InnerRng> &&
    BoundedRange<R> && BoundedRange<InnerRng>;
constexpr const_iterator end() const requires ForwardRange<const R> &&
    is_reference_v<reference_t<iterator_t<const R>>> &&
    ForwardRange<reference_t<iterator_t<const R>>> &&
    BoundedRange<const R> && BoundedRange<reference_t<iterator_t<const R>>>;
```

2 *Returns:* `{*this, ranges::end(base_)}`.

10.8.12.2 Class template `join_view::__iterator` [ranges.adaptors.join_view.iterator]

1 `join_view::__iterator` is an exposition-only type.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R>
    template <bool Const>
    struct join_view<R>::__iterator {
    private:
      using Base = conditional_t<Const, const R, R>;
      using Parent = conditional_t<Const, const join_view, join_view>;

      iterator_t<Base> outer_ {};
      iterator_t<reference_t<iterator_t<Base>>> inner_ {};
      Parent* parent_ {};

      constexpr void satisfy_();
    public:
      using iterator_category = see below;
      using value_type = value_type_t<iterator_t<reference_t<iterator_t<Base>>>>;
      using difference_type = see below;

      __iterator() = default;
      constexpr __iterator(Parent& parent, iterator_t<R> outer);
      constexpr __iterator(__iterator<!Const> i) requires Const &&
          ConvertibleTo<iterator_t<R>, iterator_t<Base>> &&
          ConvertibleTo<iterator_t<InnerRng>,
              iterator_t<reference_t<iterator_t<Base>>>>;

      constexpr decltype(auto) operator*() const;
```

```
        constexpr __iterator& operator++();
        constexpr void operator++(int);
        constexpr __iterator operator++(int)
            requires is_reference_v<reference_t<iterator_t<Base>>> &&
                ForwardRange<Base> &&
                ForwardRange<reference_t<iterator_t<Base>>>;

        constexpr __iterator& operator--();
            requires is_reference_v<reference_t<iterator_t<Base>>> &&
                BidirectionalRange<Base> &&
                BidirectionalRange<reference_t<iterator_t<Base>>>;

        constexpr __iterator operator--(int)
            requires is_reference_v<reference_t<iterator_t<Base>>> &&
                BidirectionalRange<Base> &&
                BidirectionalRange<reference_t<iterator_t<Base>>>;

        friend constexpr bool operator==(const __iterator& x, const __iterator& y)
        requires is_reference_v<reference_t<iterator_t<Base>>> &&
            EqualityComparable<iterator_t<Base>> &&
            EqualityComparable<iterator_t<reference_t<iterator_t<Base>>>>;

        friend constexpr bool operator!=(const __iterator& x, const __iterator& y)
        requires is_reference_v<reference_t<iterator_t<Base>>> &&
            EqualityComparable<iterator_t<Base>> &&
            EqualityComparable<iterator_t<reference_t<iterator_t<Base>>>>;

        friend constexpr decltype(auto) iter_move(const __iterator& i)
            noexcept(see below);

        friend constexpr void iter_swap(const __iterator& x, const __iterator& y)
            noexcept(see below);
    };
}}}}
```

2 `join_view<R>::iterator::iterator_category` is defined as follows:

(2.1) — If `Base` satisfies `BidirectionalRange`, and if `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, and if `reference_t<iterator_t<Base>>` satisfies `BidirectionalRange`, then `iterator_category` is `ranges::bidirectional_iterator_tag`.

(2.2) — Otherwise, if `Base` satisfies `ForwardRange`, and if `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, and if `reference_t<iterator_t<Base>>` satisfies `ForwardRange`, then `iterator_category` is `ranges::forward_iterator_tag`.

(2.3) — Otherwise, `iterator_category` is `ranges::input_iterator_tag`.

3 `join_view<R>::iterator::difference_type` is an alias for:

```
common_type_t<
  difference_type_t<iterator_t<Base>>,
  difference_type_t<iterator_t<reference_t<iterator_t<Base>>>>>
```

4 The `join_view<R>::iterator::satisfy_()` function is equivalent to:

```
for (; outer_ != ranges::end(parent_->base_); ++outer_) {
  auto&& inner = inner-range-update;
  inner_ = ranges::begin(inner);
  if (inner_ != ranges::end(inner))
    return;
}
if constexpr (is_reference_v<reference_t<iterator_t<Base>>>)
  inner_ = iterator_t<reference_t<iterator_t<Base>>>{};
```

where *inner-range-update* is equivalent to:

(4.1) — If `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, `*outer_`.

(4.2) — Otherwise,

```
[this](auto&& x) -> decltype(auto) {
  return (parent_->inner_ = view::all(x));
}(*outer_)
```

10.8.12.2.1 `join_view::__iterator` operations [ranges.adaptors.join_view.iterator.ops]

10.8.12.2.1.1 `join_view::__iterator` constructors [ranges.adaptors.join_view.iterator.ctor]

```
constexpr __iterator(Parent& parent, iterator_t<R> outer)
```

1 *Effects:* Initializes `outer_` with `outer` and initializes `parent_` with `&parent`; then calls `satisfy_()`.

```
constexpr __iterator(__iterator<!Const> i) requires Const &&
    ConvertibleTo<iterator_t<R>, iterator_t<Base>> &&
    ConvertibleTo<iterator_t<InnerRng>,
        iterator_t<reference_t<iterator_t<Base>>>>;
```

2 *Effects:* Initializes `outer_` with `i.outer_`, initializes `inner_` with `i.inner_`, and initializes `parent_` with `i.parent_`.

10.8.12.2.1.2 `join_view::iterator::operator*` [ranges.adaptors.join_view.iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Returns:* `*inner_`.

10.8.12.2.1.3 `join_view::iterator::operator++` [ranges.adaptors.join_view.iterator.inc]

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

1 *Effects:* Equivalent to:

```
if (++inner_ == ranges::end(inner-range)) {
  ++outer_;
  satisfy_();
}
return *this;
```

where *inner-range* is equivalent to:

(1.1) — If `is_reference_v<reference_t<iterator_t<Base>>>` is `true`, `*outer_`.

(1.2) — Otherwise, `parent_->inner_`.

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

2 *Effects:* Equivalent to:

```
(void)++*this;
```

```
constexpr __iterator operator++(int)
    requires is_reference_v<reference_t<iterator_t<Base>>> &&
        ForwardRange<Base> &&
        ForwardRange<reference_t<iterator_t<Base>>>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

10.8.12.2.1.4 `join_view::iterator::operator--` [ranges.adaptors.join_view.iterator.dec]

```
constexpr __iterator& operator--();
    requires is_reference_v<reference_t<iterator_t<Base>>> &&
        BidirectionalRange<Base> &&
        BidirectionalRange<reference_t<iterator_t<Base>>>;
```

1 *Effects:* Equivalent to:

```
if (outer_ == ranges::end(parent_->base_))
  inner_ = ranges::end(*--outer_);
while (inner_ == ranges::begin(*outer_))
  inner_ = ranges::end(*--outer_);
--inner_;
return *this;
```

```
constexpr __iterator operator--(int)
    requires is_reference_v<reference_t<iterator_t<Base>>> &&
        BidirectionalRange<Base> &&
        BidirectionalRange<reference_t<iterator_t<Base>>>;
```

2 *Effects:* Equivalent to:

```
auto tmp = *this;
--*this;
return tmp;
```

10.8.12.2.2 `join_view::__iterator` comparisons [ranges.adaptors.join_view.iterator.comp]

```
friend constexpr bool operator==(const __iterator& x, const __iterator& y)
    requires is_reference_v<reference_t<iterator_t<Base>>> &&
        EqualityComparable<iterator_t<Base>> &&
        EqualityComparable<iterator_t<reference_t<iterator_t<Base>>>>;
```

1 *Returns:* `x.outer_ == y.outer_ && x.inner_ == y.inner_`.

```
friend constexpr bool operator!=(const __iterator& x, const __iterator& y)
    requires is_reference_v<reference_t<iterator_t<Base>>> &&
        EqualityComparable<iterator_t<Base>> &&
        EqualityComparable<iterator_t<reference_t<iterator_t<Base>>>>;
```

2 *Returns:* `!(x == y)`.

10.8.12.2.3 `join_view::__iterator` non-member functions [ranges.adaptors.join_view.iterator.nonmember]

```
friend constexpr decltype(auto) iter_move(const __iterator& i)
    noexcept(see below);
```

1 *Returns:* `ranges::iter_move(i.inner_)`.

2 *Remarks:* The expression in the `noexcept` clause is equivalent to:

```
noexcept(ranges::iter_move(i.inner_))
```

```
friend constexpr void iter_swap(const __iterator& x, const __iterator& y)
    noexcept(see below);
```

3 *Returns:* `ranges::iter_swap(x.inner_, y.inner_)`.

4 *Remarks:* The expression in the `noexcept` clause is equivalent to:

```
noexcept(ranges::iter_swap(x.inner_, y.inner_))
```

10.8.12.3 Class template `join_view::__sentinel` [ranges.adaptors.join_view.sentinel]

1 `join_view::__sentinel` is an exposition-only type.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class R>
    template <bool Const>
    struct join_view<R>::__sentinel {
    private:
      using Base = conditional_t<Const, const R, R>;
      using Parent = conditional_t<Const, const join_view, join_view>;
      sentinel_t<Base> end_ {};
    public:
      __sentinel() = default;

      constexpr explicit __sentinel(Parent& parent);
      constexpr __sentinel(__sentinel<!Const> s) requires Const &&
          ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;

      friend constexpr bool operator==(const __iterator<Const>& x, const __sentinel& y);
      friend constexpr bool operator==(const __sentinel& x, const __iterator<Const>& y);
      friend constexpr bool operator!=(const __iterator<Const>& x, const __sentinel& y);
      friend constexpr bool operator!=(const __sentinel& x, const __iterator<Const>& y);
    };
}}}}
```

10.8.12.3.1 `join_view::__sentinel` operations [ranges.adaptors.join_view.sentinel.ops]

10.8.12.3.1.1 `join_view::__sentinel` constructors [ranges.adaptors.join_view.sentinel.ctor]

```
constexpr explicit __sentinel(Parent& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent.base_)`.

```
constexpr __sentinel(__sentinel<!Const> s) requires Const &&
    ConvertibleTo<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes `end_` with `s.end_`.

10.8.12.3.2 `join_view::__sentinel` comparisons [ranges.adaptors.join_view.sentinel.comp]

```
friend constexpr bool operator==(const __iterator<Const>& x, const __sentinel& y);
```

1 *Returns:* `x.outer_ == y.end_`.

```
friend constexpr bool operator==(const __sentinel& x, const __iterator<Const>& y);
```

2 *Returns:* `y == x`.

```
friend constexpr bool operator!=(const __iterator<Const>& x, const __sentinel& y);
```

3 *Returns:* `!(x == y)`.

```
friend constexpr bool operator!=(const __sentinel& x, const __iterator<Const>& y);
```

4 *Returns:* `!(y == x)`.

10.8.13 `view::join` [ranges.adaptors.join]

1 The name `view::join` denotes a range adaptor object (10.8.1). Let `E` be an expression such that type `T` is `decltype((E))`. Then the expression `view::join(E)` is expression-equivalent to:

(1.1) — `join_view{E}` if the following is satisfied:

```
InputRange<T> &&
InputRange<reference_t<iterator_t<T>>> &&
(is_reference_v<reference_t<iterator_t<T>>> ||
 View<value_type_t<iterator_t<T>>)
```

(1.2) — Otherwise, `view::join(E)` is ill-formed.

10.8.14 Class template `empty_view` [ranges.adaptors.empty_view]

1 The purpose of `empty_view` is to produce an emtpy range of elements of a particular type.

2 [*Example:*

```
empty_view<int> e;
static_assert(ranges::empty(e));
static_assert(0 == e.size());
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class T>
    requires requires { *(T*)nullptr; }
  class empty_view : view_interface<empty_view<T>> {
  public:
    empty_view() = default;

    using iterator = T*;
    using const_iterator = T*;
    using sentinel = T*;
    using const_sentinel = T*;

    constexpr static T* begin() noexcept;
    constexpr static T* end() noexcept;
    constexpr static ptrdiff_t size() noexcept;
    constexpr static T* data() noexcept;
  };
}}}}
```

10.8.14.1 `empty_view` operations [ranges.adaptors.empty_view.ops]

10.8.14.1.1 `empty_view` begin [ranges.adaptors.empty_view.begin]

```
constexpr static T* begin() noexcept;
```

1 *Returns:* `nullptr`.

10.8.14.1.2 `empty_view` end [ranges.adaptors.empty_view.end]

```
constexpr static T* end() noexcept;
```

1 *Returns:* `nullptr`.

10.8.14.1.3 `empty_view` size [ranges.adaptors.empty_view.size]

```
constexpr static ptrdiff_t size() noexcept;
```

1 *Returns:* `0`.

10.8.14.1.4 `empty_view` data [ranges.adaptors.empty_view.data]

```
constexpr static T* data() noexcept;
```

1 *Returns:* `nullptr`.

10.8.15 Class template `single_view` [ranges.adaptors.single_view]

1 The purpose of `single_view` is to produce a range that contains exactly one element of a specified value.

2 [*Example:*

```
single_view s{4};
for (int i : s)
  cout << i; // prints 4
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <CopyConstructible T>
  class single_view : view_interface<single_view<T>> {
  private:
    semiregular<T> value_; // exposition only
  public:
    single_view() = default;
    constexpr explicit single_view(const T& t);
    constexpr explicit single_view(T&& t);
    template <class... Args>
      requires Constructible<T, Args...>
    constexpr single_view(in_place_t, Args&&... args);

    using iterator = const T*;
    using const_iterator = const T*;
    using sentinel = const T*;
    using const_sentinel = const T*;

    constexpr const T* begin() const noexcept;
    constexpr const T* end() const noexcept;
    constexpr static ptrdiff_t size() noexcept;
    constexpr const T* data() const noexcept;
  };
```

```
    template <class T>
    requires CopyConstructible<decay_t<T>>
    explicit single_view(T&&) -> single_view<decay_t<T>>;
  }}}}
```

10.8.15.1 `single_view` operations [ranges.adaptors.single_view.ops]

10.8.15.1.1 `single_view` constructors [ranges.adaptors.single_view.ctor]

```
constexpr explicit single_view(const T& t);
```

1 *Effects:* Initializes `value_` with `t`.

```
constexpr explicit single_view(T&& t);
```

2 *Effects:* Initializes `value_` with `std::move(t)`.

```
template <class... Args>
constexpr single_view(in_place_t, Args&&... args);
```

3 *Effects:* Initializes `value_` as if by `value_{in_place, std::forward<Args>(args)...}`.

10.8.15.1.2 `single_view` begin [ranges.adaptors.single_view.begin]

```
constexpr const T* begin() const noexcept;
```

1 *Requires:* `bool(value_)`

2 *Returns:* `value_.operator->()`.

10.8.15.1.3 `single_view` end [ranges.adaptors.single_view.end]

```
constexpr const T* end() const noexcept;
```

1 *Requires:* `bool(value_)`

2 *Returns:* `value_.operator->() + 1`.

10.8.15.1.4 `single_view` size [ranges.adaptors.single_view.size]

```
constexpr static ptrdiff_t size() noexcept;
```

1 *Requires:* `bool(value_)`

2 *Returns:* `1`.

10.8.15.1.5 `single_view` data [ranges.adaptors.single_view.data]

```
constexpr const T* data() const noexcept;
```

1 *Requires:* `bool(value_)`

2 *Returns:* `begin()`.

10.8.16 `view::single` [ranges.adaptors.single]

1 The name `view::single` denotes a customization point object (). Let `E` be an expression such that its un-*cv* qualified type is `I`. Then the expression `view::single(E)` is expression-equivalent to:

(1.1) — `single_view{E}` if `CopyConstructible<I>` is satisfied.

(1.2) — Otherwise, `view::single(E)` is ill-formed.

10.8.17 Class template `split_view` [ranges.adaptors.split_view]

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

2 [*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*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // exposition only
  template <class R>
  concept bool tiny-range =
    SizedRange<R> && requires {
      requires remove_reference_t<R>::size() <= 1;
    };

  template <InputRange Rng, ForwardRange Pattern>
    requires View<Rng> && View<Pattern> &&
        IndirectlyComparable<iterator_t<Rng>, iterator_t<Pattern>> &&
        (ForwardRange<Rng> || tiny-range<Pattern>)
  struct split_view {
  private:
    Rng base_ {}; // expos
    Pattern pattern_ {}; // expos
    iterator_t<Rng> current_ {}; // expos, only present if !ForwardRange<Rng>
    template <bool Const> struct __outer_iterator; // expos
    template <bool Const> struct __outer_sentinel; // expos
    template <bool Const> struct __inner_iterator; // expos
    template <bool Const> struct __inner_sentinel; // expos
  public:
    split_view() = default;
    constexpr split_view(Rng base, Pattern pattern);

    template <InputRange O, ForwardRange P>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
          (is_lvalue_reference_v<P> || View<decay_t<P>>) &&
          Constructible<Rng, all_view<O>> &&
          Constructible<Pattern, all_view<P>>
    constexpr split_view(O&& o, P&& p);

    template <InputRange O>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
          Constructible<Rng, all_view<O>> &&
          Constructible<Pattern, single_view<value_type_t<iterator_t<O>>>>
    constexpr split_view(O&& o, value_type_t<iterator_t<O>> e);

    using iterator = __outer_iterator<false>;
```

```
      using sentinel = __outer_sentinel<false>;
      using const_iterator = __outer_iterator<true>;
      using const_sentinel = __outer_sentinel<true>;

      constexpr iterator begin();
      constexpr iterator begin() requires ForwardRange<Rng>;
      constexpr const_iterator begin() const requires ForwardRange<Rng>;

      constexpr sentinel end()
      constexpr const_sentinel end() const requires ForwardRange<Rng>;

      constexpr iterator end()
        requires ForwardRange<Rng> && BoundedRange<Rng>;
      constexpr const_iterator end() const
        requires ForwardRange<Rng> && BoundedRange<Rng>;
    };

    template <InputRange O, ForwardRange P>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
        (is_lvalue_reference_v<P> || View<decay_t<P>>) &&
        IndirectlyComparable<iterator_t<O>, iterator_t<P>> &&
        (ForwardRange<O> || _TinyRange<P>)
    split_view(O&&, P&&) -> split_view<all_view<O>, all_view<P>>;

    template <InputRange O>
      requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
        IndirectlyComparable<iterator_t<Rng>, const value_type_t<iterator_t<Rng>>*> &&
        CopyConstructible<value_type_t<iterator_t<O>>>
    split_view(O&&, value_type_t<iterator_t<O>>)
      -> split_view<all_view<O>, single_view<value_type_t<iterator_t<O>>>>;
  }}}}
```

10.8.17.1 `split_view` operations [ranges.adaptors.split_view.ops]

10.8.17.1.1 `split_view` constructors [ranges.adaptors.split_view.ctor]

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

1 *Effects:* Initializes `base_` with `std::move(base)` and initializes `pattern_` with `std::move(pattern)`.

```
template <InputRange O, ForwardRange P>
  requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
      (is_lvalue_reference_v<P> || View<decay_t<P>>) &&
      Constructible<Rng, all_view<O>> &&
      Constructible<Pattern, all_view<P>>
constexpr split_view(O&& o, P&& p);
```

2 *Effects:* Delegates to `split_view{view::all(std::forward<O>(o)), view::all(std::forward<P>(p))}`.

```
template <InputRange O>
  requires (is_lvalue_reference_v<O> || View<decay_t<O>>) &&
      Constructible<Rng, all_view<O>> &&
      Constructible<Pattern, single_view<value_type_t<iterator_t<O>>>>
constexpr split_view(O&& o, value_type_t<iterator_t<O>> e);
```

3 *Effects:* Delegates to `split_view{view::all(std::forward<O>(o)), single_view{std::move(e)}}`.

10.8.17.1.2 split_view range begin [ranges.adaptors.split_view.begin]

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

1 *Effects:* Equivalent to:

```
current_ = ranges::begin(base_);
return iterator{*this};
```

```
constexpr iterator begin() requires ForwardRange<Rng>;
constexpr const_iterator begin() const requires ForwardRange<Rng>;
```

2 *Returns:* `{*this, ranges::begin(base_)}`.

10.8.17.1.3 split_view range end [ranges.adaptors.split_view.end]

```
constexpr sentinel end()
constexpr const_sentinel end() const requires ForwardRange<Rng>;
```

1 *Effects:* Equivalent to `sentinel{*this}` and `const_sentinel{*this}` for the first and second overloads, respectively.

```
constexpr iterator end()
  requires ForwardRange<Rng> && BoundedRange<Rng>;
constexpr const_iterator end() const
  requires ForwardRange<Rng> && BoundedRange<Rng>;
```

2 *Returns:* `{*this, ranges::end(base_)}`.

10.8.17.2 Class template split_view::__outer_iterator [ranges.adaptors.split_view.outer_iterator]

1 [*Note:* `split_view::__outer_iterator` is an exposition-only type. — *end note*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Rng, class Pattern>
  template <bool Const>
  struct split_view<Rng, Pattern>::__outer_iterator {
  private:
    using Base = conditional_t<Const, const Rng, Rng>;
    using Parent = conditional_t<Const, const split_view, split_view>;
    iterator_t<Base> current_ {}; // Only present if ForwardRange<Rng> is satisfied
    Parent* parent_ = nullptr;
  public:
    using iterator_category = see below;
    using difference_type = difference_type_t<iterator_t<Base>>;
    struct value_type;

    __outer_iterator() = default;
    constexpr explicit __outer_iterator(Parent& parent);
    constexpr __outer_iterator(Parent& parent, iterator_t<Base> current)
      requires ForwardRange<Base>;
    constexpr __outer_iterator(__outer_iterator<!Const> i) requires Const &&
      ConvertibleTo<iterator_t<Rng>, iterator_t<Base>>;

    constexpr value_type operator*() const;

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

```
    constexpr __outer_iterator operator++(int) requires ForwardRange<Base>;

    friend constexpr bool operator==(const __outer_iterator& x, const __outer_iterator& y)
      requires ForwardRange<Base>;
    friend constexpr bool operator!=(const __outer_iterator& x, const __outer_iterator& y)
      requires ForwardRange<Base>;
  };
}}}}
```

2 `split_view<Rng, Pattern>::__outer_iterator::iterator_category` is defines as follows:

(2.1) — If `__outer_iterator::Base` satisfies `ForwardRange`, then `iterator_category` is `ranges::forward_iterator_tag`.

(2.2) — Otherwise, `iterator_category` is `ranges::input_iterator_tag`.

10.8.17.3 `split_view::__outer_iterator` operations [ranges.adaptors.split_view.outer_iterator.ops]

10.8.17.3.1 `split_view::__outer_iterator` constructors [ranges.adaptors.split_view.outer_iterator.ctor]

```
constexpr explicit __outer_iterator(Parent& parent);
```

1 *Effects:* Initializes `parent_` with `&parent`.

```
constexpr __outer_iterator(Parent& parent, iterator_t<Base> current)
  requires ForwardRange<Base>;
```

2 *Effects:* Initializes `parent_` with `&parent` and `current_` with `current`.

```
constexpr __outer_iterator(__outer_iterator<!Const> i) requires Const &&
  ConvertibleTo<iterator_t<Rng>, iterator_t<Base>>;
```

3 *Effects:* Initializes `parent_` with `i.parent_` and `current_` with `i.current_`.

10.8.17.3.2 `split_view::__outer_iterator::operator*` [ranges.adaptors.split_view.outer_iterator.star]

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

1 *Returns:* `value_type{*this}`.

10.8.17.3.3 `split_view::__outer_iterator::operator++` [ranges.adaptors.split_view.outer_iterator.inc]

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

1 *Effects:* Equivalent to:

```
auto const end = ranges::end(parent_->base_);
if (current == end) return *this;
auto const [pbegin, pend] = iterator_range{parent_->pattern_};
do {
  auto [b,p] = mismatch(current, end, pbegin, pend);
  if (p != pend) continue; // The pattern didn't match
  current = bump(b, pbegin, pend, end); // skip the pattern
  break;
} while (++current != end);
return *this;
```

Where *current* is equivalent to:

(1.1) — If `Rng` satisfies `ForwardRange`, `current_`.

(1.2) — Otherwise, `parent_->current_`.

and *bump*`(b, x, y, e)` is equivalent to:

(1.3) — If `Rng` satisfies `ForwardRange`, `next(b, (int)(x == y), e)`.

(1.4) — Otherwise, `b`.

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

2 *Effects:* Equivalent to `(void)++*this`.

```
constexpr __outer_iterator operator++(int) requires ForwardRange<Base>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

10.8.17.3.4 `split_view::__outer_iterator` non-member functions [ranges.adaptors.split__view.outer__iterator.nonmember]

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

1 *Effects:* Equivalent to:

```
return x.current_ == y.current_;
```

```
friend constexpr bool operator!=(const __outer_iterator& x, const __outer_iterator& y)
  requires ForwardRange<Base>;
```

2 *Effects:* Equivalent to:

```
return !(x == y);
```

10.8.17.4 Class template `split_view::__outer_sentinel` [ranges.adaptors.split__view.outer__sentinel]

1 [*Note:* `split_view::__outer_sentinel` is an exposition-only type. — *end note*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Rng, class Pattern>
  template <bool Const>
  struct split_view<Rng, Pattern>::__outer_sentinel {
  private:
    using Base = conditional_t<Const, const Rng, Rng>;
    using Parent = conditional_t<Const, const split_view, split_view>;
    sentinel_t<Base> end_;
  public:
    __outer_sentinel() = default;
    constexpr explicit __outer_sentinel(Parent& parent);
```

```
    friend constexpr bool operator==(const __outer_iterator<Const>& x, const __outer_sentinel& y);
    friend constexpr bool operator==(const __outer_sentinel& x, const __outer_iterator<Const>& y);
    friend constexpr bool operator!=(const __outer_iterator<Const>& x, const __outer_sentinel& y);
    friend constexpr bool operator!=(const __outer_sentinel& x, const __outer_iterator<Const>& y);
  };
}}}}
```

10.8.17.4.1 `split_view::__outer_sentinel` constructors [ranges.adaptors.split__view.outer__sentinel.ctor]

```
constexpr explicit __outer_sentinel(Parent& parent);
```

1 *Effects:* Initializes `end_` with `ranges::end(parent.base_)`.

10.8.17.4.2 `split_view::__outer_sentinel` non-member functions [ranges.adaptors.split__view.outer__sentinel.nonmember]

```
friend constexpr bool operator==(const __outer_iterator<Const>& x, const __outer_sentinel& y);
```

1 *Effects:* Equivalent to:

```
return current(x) == y.end_;
```

Where *`current`*`(x)` is equivalent to:

(1.1) — If `Rng` satisfies `ForwardRange`, `x.current_`.

(1.2) — Otherwise, `x.parent_->current_`.

```
friend constexpr bool operator==(const __outer_sentinel& x, const __outer_iterator<Const>& y);
```

2 *Effects:* Equivalent to:

```
return y == x;
```

```
friend constexpr bool operator!=(const __outer_iterator<Const>& x, const __outer_sentinel& y);
```

3 *Effects:* Equivalent to:

```
return !(x == y);
```

```
friend constexpr bool operator!=(const __outer_sentinel& x, const __outer_iterator<Const>& y);
```

4 *Effects:* Equivalent to:

```
return !(y == x);
```

10.8.17.5 Class `split_view::__outer_iterator::value_type` [ranges.adaptors.split__view.outer__iterator.value__type]

1 [*Note:* `split_view::__outer_iterator::value_type` is an exposition-only type. — *end note*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Rng, class Pattern>
  template <bool Const>
  struct split_view<Rng, Pattern>::__outer_iterator<Const>::value_type {
  private:
    __outer_iterator i_ {};
  public:
    value_type() = default;
```

```
      constexpr explicit value_type(__outer_iterator i);

      using iterator = __inner_iterator<Const>;
      using sentinel = __inner_sentinel<Const>;
      using const_iterator = __inner_iterator<Const>;
      using const_sentinel = __inner_sentinel<Const>;

      constexpr iterator begin() const;
      constexpr sentinel end() const;
    };
  }}}}
```

10.8.17.5.1 split_view::__outer_iterator::value_type constructors [ranges.adaptors.split_view.outer_iterator.value_type.ctor]

```
constexpr explicit value_type(__outer_iterator i);
```

1 *Effects:* Initializes `i_` with `i`.

10.8.17.5.2 split_view::__outer_iterator::value_type range begin [ranges.adaptors.split_view.outer_iterator.value_type.begin]

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

1 *Returns:* `iterator{i_}`.

10.8.17.5.3 split_view::__outer_iterator::value_type range end [ranges.adaptors.split_view.outer_iterator.value_type.end]

```
constexpr sentinel end() const;
```

1 *Returns:* `sentinel{}`.

10.8.17.6 Class template split_view::__inner_iterator [ranges.adaptors.split_view.inner_iterator]

1 [*Note:* `split_view::__inner_iterator` is an exposition-only type. — *end note*]

2 In the definition of `split_view<Rng, Pattern>::__inner_iterator` below, *`current`*`(i)` is equivalent to:

(2.1) — If `Rng` satisfies `ForwardRange`, `i.current_`.

(2.2) — Otherwise, `i.parent_->current_`.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Rng, class Pattern>
  template <bool Const>
  struct split_view<Rng, Pattern>::__inner_iterator {
  private:
    using Base = conditional_t<Const, const Rng, Rng>;
    __outer_iterator<Const> i_ {};
    bool zero_ = false;
  public:
    using iterator_category = iterator_category_t<__outer_iterator<Const>>;
    using difference_type = difference_type_t<iterator_t<Base>>;
    using value_type = value_type_t<iterator_t<Base>>;

    __inner_iterator() = default;
```

```
      constexpr explicit __inner_iterator(__outer_iterator<Const> i);

      constexpr decltype(auto) operator*() const;

      constexpr __inner_iterator& operator++();
      constexpr void operator++(int);
      constexpr __inner_iterator operator++(int) requires ForwardRange<Base>;

      friend constexpr bool operator==(const __inner_iterator& x, const __inner_iterator& y)
        requires ForwardRange<Base>;
      friend constexpr bool operator!=(const __inner_iterator& x, const __inner_iterator& y)
        requires ForwardRange<Base>;

      friend constexpr decltype(auto) iter_move(const __inner_iterator& i)
        noexcept(see below);
      friend constexpr void iter_swap(const __inner_iterator& x, const __inner_iterator& y)
        noexcept(see below) requires IndirectlySwappable<iterator_t<Base>>;
    };
  }}}}
```

10.8.17.6.1 `split_view::__inner_iterator` constructors [ranges.adaptors.split_view.inner_iterator.ctor]

```
constexpr explicit __inner_iterator(__outer_iterator<Const> i);
```

1 *Effects:* Initializes `i_` with `i`.

10.8.17.6.2 `split_view::__inner_iterator::operator*` [ranges.adaptors.split_view.inner_iterator.star]

```
constexpr decltype(auto) operator*() const;
```

1 *Returns:* `*current(i_)`.

10.8.17.6.3 `split_view::__inner_iterator::operator++` [ranges.adaptors.split_view.inner_iterator.inc]

```
constexpr decltype(auto) operator++() const;
```

1 *Effects:* Equivalent to:

```
++current(i_);
zero_ = true;
return *this;
```

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

2 *Effects:* Equivalent to `(void)++*this`.

```
constexpr __inner_iterator operator++(int) requires ForwardRange<Base>;
```

3 *Effects:* Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

10.8.17.6.4 `split_view::__inner_iterator` comparisons [ranges.adaptors.split_view.inner_iterator.comp]

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

1 *Effects:* Equivalent to:

```
return x.i_.current_ == y.i_.current_;
```

```
friend constexpr bool operator!=(const __inner_iterator& x, const __inner_iterator& y)
  requires ForwardRange<Base>;
```

2 *Effects:* Equivalent to:

```
return !(x == y);
```

10.8.17.6.5 `split_view::__inner_iterator` non-member functions [ranges.adaptors.split_view.inner_iterator.nonmember]

```
friend constexpr decltype(auto) iter_move(const __inner_iterator& i)
  noexcept(see below);
```

1 *Returns:* `ranges::iter_move(`*`current`*`(i.i_))`.

2 *Remarks:* The expression in the `noexcept` clause is equivalent to:

```
noexcept(ranges::iter_move(current(i.i_)))
```

```
friend constexpr void iter_swap(const __inner_iterator& x, const __inner_iterator& y)
  noexcept(see below) requires IndirectlySwappable<iterator_t<Base>>;
```

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

4 *Remarks:* The expression in the `noexcept` clause is equivalent to:

```
noexcept(ranges::iter_swap(current(x.i_), current(y.i_)))
```

10.8.17.7 Class template `split_view::__inner_sentinel` [ranges.adaptors.split_view.inner_sentinel]

1 [*Note:* `split_view::__inner_sentinel` is an exposition-only type. — *end note*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Rng, class Pattern>
  template <bool Const>
  struct split_view<Rng, Pattern>::__inner_sentinel {
    friend constexpr bool operator==(const __inner_iterator<Const>& x, __inner_sentinel);
    friend constexpr bool operator==(__inner_sentinel x, const __inner_iterator<Const>& y);
    friend constexpr bool operator!=(const __inner_iterator<Const>& x, __inner_sentinel y);
    friend constexpr bool operator!=(__inner_sentinel x, const __inner_iterator<Const>& y);
  };
}}}}
```

10.8.17.7.1 `split_view::__inner_sentinel` comparisons [ranges.adaptors.split__view.inner__sentinel.comp]

```
friend constexpr bool operator==(const __inner_iterator<Const>& x, __inner_sentinel)
```

1 *Effects:* Equivalent to:

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

```
friend constexpr bool operator==(__inner_sentinel x, const __inner_iterator<Const>& y);
```

2 *Effects:* Equivalent to:

```
return y == x;
```

```
friend constexpr bool operator!=(const __inner_iterator<Const>& x, __inner_sentinel y);
```

3 *Effects:* Equivalent to:

```
return !(x == y);
```

```
friend constexpr bool operator!=(__inner_sentinel x, const __inner_iterator<Const>& y);
```

4 *Effects:* Equivalent to:

```
return !(y == x);
```

10.8.18 `view::split` [ranges.adaptors.split]

1 The name `view::split` denotes a range adaptor object (). Let `E` and `F` be expressions such that their types are `T` and `U` respectively. Then the expression `view::split(E, F)` is expression-equivalent to:

(1.1) — `split_view{E, F}` if either of the following sets of requirements is satisfied:

(1.1.1) — `InputRange<T> && ForwardRange<U> &&`
`(is_lvalue_reference_v<T> || View<decay_t<T>>) &&`
`(is_lvalue_reference_v<U> || View<decay_t<U>>) &&`
`IndirectlyComparable<iterator_t<T>, iterator_t<U>> &&`
`(ForwardRange<T> || `*`tiny-range`*`<U>)`

(1.1.2) — `InputRange<T> && (is_lvalue_reference_v<T> || View<decay_t<T>>) &&`
`IndirectlyComparable<iterator_t<T>, const value_type_t<iterator_t<T>>*> &&`
`CopyConstructible<value_type_t<iterator_t<T>>> &&`
`ConvertibleTo<U, value_type_t<iterator_t<T>>>`

(1.2) — Otherwise, `view::split(E, F)` is ill-formed.

10.8.19 `view::counted` [ranges.adaptors.counted]

1 The name `view::counted` denotes a customization point object (). Let `E` and `F` be expressions such that their decayed types are `T` and `U` respectively. Then the expression `view::counted(E, F)` is expression-equivalent to:

(1.1) — `iterator_range{E, E + F}` is `T` is a pointer to an object type, and if `U` is implicitly convertible to `ptrdiff_t`.

(1.2) — Otherwise, `iterator_range{make_counted_iterator(E, F), default_sentinel{}}` if `Iterator<T> && ConvertibleTo<U, difference_type_t<T>>` is satisfied.

(1.3) — Otherwise, `view::counted(E, F)` is ill-formed.

Annex A (informative) Acknowledgements [acknowledgements]

This work was made possible in part by a grant from the Standard C++ Foundation, and by the support of my employer, Facebook.

Bibliography

[1] Eric Niebler and Casey Carter. N4685: C++ extensions for ranges, 7 2017. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/n4685.pdf.

[2] Eric Niebler, Sean Parent, and Andrew Sutton. N4128: Ranges for the standard library, revision 1, 10 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html.

Index

Index of library names