Document Number: P0789R1
Date: 2017-11-17
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" ([?]).]

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

(1.1) — ~~An `iterator_range`~~A `subrange` 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::empty` 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 or the Ranges TS 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 {
  // 10.6.11:
  template <class T>
  concept bool ViewableRange = see below;

  // 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;

  enum class subrange_kind : bool { unsized, sized };

  // 10.7.2.1:
  template <Iterator I, Sentinel<I> S = I, subrange_kind K = see below>
    requires K == subrange_kind::sized || !SizedSentinel<S, I>
  class subrange;

  // 10.8.4:
  namespace view { inline constexpr unspecified  all = unspecified ; }

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

  // 10.8.5:
  template <InputRange R, IndirectUnaryPredicate<iterator_t<R>> Pred>
    requires View<R>
  class filter_view;

  namespace view { inline constexpr unspecified  filter = unspecified ; }

  // 10.8.7:
  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 ; }

  // 10.8.9:
  template <WeaklyIncrementable I, Semiregular Bound = unreachable>
```

```
  requires WeaklyEqualityComparable<I, Bound>
class iota_view;

namespace view { inline constexpr unspecified  iota = unspecified ; }

// 10.8.13:
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 ; }

// 10.8.15:
template <class T>
  requires ~~requires { *(T*)nullptr; }~~is_object_v<T>
class empty_view;

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

// 10.8.16:
template <CopyConstructible T>
class single_view;

namespace view { inline constexpr unspecified  single = unspecified ; }

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

// 10.8.18:
template <InputRange Rng, ForwardRange Pattern>
  requires View<Rng> && View<Pattern> &&
      IndirectlyComparable<iterator_t<Rng>, iterator_t<Pattern>> &&
      (ForwardRange<Rng> || tiny-range<Pattern>)
~~struct~~class split_view;

namespace view { inline constexpr unspecified  split = unspecified ; }

// 10.8.20:
namespace view { inline constexpr unspecified  counted = unspecified ; }

// 10.8.21:
template <View Rng>
  requires !BoundedRange<Rng>
class bounded_view;

namespace view { inline constexpr unspecified bounded = unspecified; }

// 10.8.23:
template <View Rng>
```

```
    requires BidirectionalRange<Rng>
  class reverse_view;

  namespace view { inline constexpr unspecified reverse = unspecified; }
}}}}

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

10.6.11 Viewable ranges [ranges.viewable]

1 The `ViewableRange` concept specifies the requirements of a `Range` type that can be converted to a `View` safely.

```
template <class T>
concept bool ViewableRange =
  Range<T> && (is_lvalue_reference_v<T> || View<decay_t<T>>); // see below
```

2 There need not be any subsumption relationship between `ViewableRange<T>` and `is_lvalue_reference_v<T>`.

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 : public 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>~~requires { ranges::empty(derived()); };
    ~~constexpr bool operator!() const requires ForwardRange<const D>;~~

    template <RandomAccessRange R = const D>
      constexpr auto data() const requires is_pointer_v<iterator_t<R>;

    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_treference_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>requires { ranges::empty(derived()); };
```

2 ~~*Returns:* !empty().~~ *Effects:* Equivalent to: `return !ranges::empty(derived());`

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

3 *Returns:* `empty()`.

```
template <RandomAccessRange R = const D>
  constexpr auto data() const requires is_pointer_v<iterator_t<R>>;
```

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

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

5 *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>;
```

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

7 *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>;
```

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

9 *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;
```

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

11 *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;
```

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

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

```
template <ForwardRange C>
  requires !View<C> && ~~MoveConstructible<C> &&~~
    ConvertibleTo<~~value_type_t~~reference_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;
```

14 *Effects:* Equivalent to:

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

10.7.2 Sub-ranges [ranges.subranges]

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.~~The `subrange` class template bundles together an iterator and a sentinel into a single object that satisfies the `View` concept. Additionally, it satisfies the `SizedRange` concept when the final template parameter is `subrange`_kind::sized.

[Editor's note: Remove sections [ranges.iterator_range] and [ranges.sized_iterator_range] and replace with the following:]

10.7.2.1 `subrange` [ranges.subrange]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class T>
  concept bool pair-like = // exposition only
    requires(T t) {
      { tuple_size<T>::value } -> Integral;
      requires tuple_size<T>::value == 2;
      typename tuple_element_t<0, T>;
      typename tuple_element_t<1, T>;
      { get<0>(t) } -> const tuple_element_t<0, T>&;
      { get<1>(t) } -> const tuple_element_t<1, T>&;
    };

  template <class T, class U, class V>
  concept bool pair-like-convertible-to = // exposition only
    !Range<T> && pair-like<decay_t<T>> &&
    requires(T&& t) {
      { get<0>(std::forward<T>(t)) } -> ConvertibleTo<U>;
      { get<1>(std::forward<T>(t)) } -> ConvertibleTo<V>;
    };
```

```
template <class T, class U, class V>
concept bool pair-like-convertible-from = // exposition only
  !Range<T> && Same<T, decay_t<T>> && pair-like<T> &&
  Constructible<T, U, V>;

template <class T>
concept bool iterator-sentinel-pair = // exposition only
  !Range<T> && Same<T, decay_t<T>> && pair-like<T> &&
  Sentinel<tuple_element_t<1, T>, tuple_element_t<0, T>>;

template <Iterator I, Sentinel<I> S = I, subrange_kind K = see below>
  requires K == subrange_kind::sized || !SizedSentinel<S, I>
class subrange : public view_interface<subrange<I, S, K>> {
private:
  static constexpr bool StoreSize =
    K == subrange_kind::sized && !SizedSentinel<S, I>; // exposition only
  I begin_ {}; // exposition only
  S end_ {}; // exposition only
  difference_type_t<I> size_ = 0; // exposition only; only present when StoreSize is true
public:
  using iterator = I;
  using sentinel = S;

  subrange() = default;

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

  constexpr subrange(I i, S s, difference_type_t<I> n)
    requires K == subrange_kind::sized;

  template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
  constexpr subrange(subrange<X, Y, Z> r)
    requires !StoreSize || Z == subrange_kind::sized;

  template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
  constexpr subrange(subrange<X, Y, Z> r, difference_type_t<I> n)
    requires K == subrange_kind::sized;

  template <pair-like-convertible-to<I, S> PairLike>
  constexpr subrange(PairLike&& r) requires !StoreSize;

  template <pair-like-convertible-to<I, S> PairLike>
  constexpr subrange(PairLike&& r, difference_type_t<I> n)
    requires K == subrange_kind::sized;

  template <Range R>
    requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
  constexpr subrange(R& r) requires !StoreSize || SizedRange<R>;

  template <pair-like-convertible-from<const I&, const S&> PairLike>
  constexpr operator PairLike() const;

  constexpr I begin() const;
  constexpr S end() const;
  constexpr bool empty() const;
```

```
    constexpr difference_type_t<I> size() const
      requires K == subrange_kind::sized;
    [[nodiscard]] constexpr subrange next(difference_type_t<I> n = 1) const;
    [[nodiscard]] constexpr subrange prev(difference_type_t<I> n = 1) const
      requires BidirectionalIterator<I>;
    constexpr subrange& advance(difference_type_t<I> n);
  };

  template <Iterator I, Sentinel<I> S>
  subrange(I, S, difference_type_t<I>) -> subrange<I, S, subrange_kind::sized>;

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

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

  template <Iterator I, Sentinel<I> S, subrange_kind K>
  subrange(subrange<I, S, K>, difference_type_t<I>) ->
    subrange<I, S, subrange_kind::sized>;

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

  template <SizedRange R>
  subrange(R&) -> subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

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

1 The default value for `subrange`'s third (non-type) template parameter is:

(1.1) — If `SizedSentinel<S, I>` is satisfied, `subrange_kind::sized`.

(1.2) — Otherwise, `subrange_kind::unsized`.

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

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

1 *Effects:* Initializes `begin_` with `i` and `end_` with `s`.

```
constexpr subrange(I i, S s, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

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

3 *Effects:* Initializes `begin_` with `i`, `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `n`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
constexpr subrange(subrange<X, Y, Z> r)
  requires !StoreSize || Z == subrange_kind::sized;
```

4 *Effects:* Equivalent to:

(4.1) — If `StoreSize` is `true`, `subrange{r.begin(), r.end(), r.size()}`.

(4.2) — Otherwise, `subrange{r.begin(), r.end()}`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
constexpr subrange(subrange<X, Y, Z> r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

5 *Effects:* Equivalent to `subrange{r.begin(), r.end(), n}`.

```
template <pair-like-convertible-to<I, S> PairLike>
constexpr subrange(PairLike&& r) requires !StoreSize;
```

6 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r))}
```

```
template <pair-like-convertible-to<I, S> PairLike>
constexpr subrange(PairLike&& r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

7 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r)), n}
```

```
template <Range R>
  requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R& r) requires !StoreSize || SizedRange<R>;
```

8 *Effects:* Equivalent to:

(8.1) — If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), distance(r)}`.

(8.2) — Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

10.7.2.1.2 subrange operators [ranges.subrange.ops]

```
template <pair-like-convertible-from<const I&, const S&> PairLike>
constexpr operator PairLike() const;
```

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

10.7.2.1.3 subrange accessors [ranges.subrange.accessors]

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

1 *Effects:* Equivalent to: `return begin_;`.

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

2 *Effects:* Equivalent to: `return end_;`.

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

3 *Effects:* Equivalent to: `return begin_ == end_;`.

```
constexpr difference_type_t<I> size() const
  requires K == subrange_kind::sized;
```

4 *Effects:* Equivalent to:

(4.1) — It `StoreSize` is `true`, `return size_;`.

(4.2) — Otherwise, `return end_ - begin_;`.

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

5 *Effects:* Equivalent to:

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

6 [*Note:* If `ForwardIterator<I>` is not satisfied, `next` may invalidate `*this`. — *end note*]

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

7 *Effects:* Equivalent to:

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

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

8 *Effects:* Equivalent to:

(8.1) — If `StoreSize` is `true`,

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

(8.2) — Otherwise,

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

10.7.2.1.4 subrange non-member functions [ranges.subrange.nonmember]

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

1 *Effects:* Equivalent to:

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

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 closure object* is a unary function object that accepts a `ViewableRange` as an argument and returns a `View`. For a range adaptor closure object `C` and an expression `R` such that `decltype((R))` satisfies `ViewableRange`, the following expressions are equivalent and return a `View`:

```
C(R)
R | C
```

Given an additional range adaptor closure objects `D`, the expression `C | D` is well-formed and produces another range adaptor closure object such that the following two expressions are equivalent:

```
R | C | D
R | (C | D)
```

2 A *range adaptor object* is a customization point object () that accepts a ~~`Range`~~`ViewableRange` as its first argument and ~~that~~ returns a `View`.

3 If the adaptor accepts only one argument, then ~~the following alternate syntaxes are semantically equivalent:~~it is a range adaptor closure object.

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

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

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

In this case, `adaptor(args...)` is a range adaptor closure object.

5 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 Simple views [ranges.adaptor.simple_view]

1 Many of the types in this section are specified in terms of an exposition-only Boolean variable template called *simple-view*<T>, defined as follows:

```
template <class R>
concept bool __simple-view =
  View<R> && View<const R> &&
  Same<iterator_t<R>, iterator_t<const R>> &&
  Same<sentinel_t<R>, sentinel_t<const R>>;

template <class R>
  constexpr bool simple-view = false;

template <__simple-view R>
  constexpr bool simple-view<R> = true;
```

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

(2.1) — *DECAY_COPY*`(E)` if the decayed 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}`~~`subrange{E}` if `E` is an lvalue and has a type that satisfies concept `Range`.

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

Remark: Whenever `view::all(E)` is a valid expression, it is a prvalue whose type satisfies `View`.

10.8.5 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 : public 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>>)ViewableRange<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>>)ViewableRange<R>
  filter_view(R&&, Pred) -> filter_view<all_view<R>, Pred>;
}}}}
```

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

10.8.5.1.1 `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>>)ViewableRange<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)`.

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

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

1 *Returns:* `base_`.

10.8.5.1.3 `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.

10.8.5.1.4 `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_)}`.

10.8.5.2 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`.

10.8.5.2.1 `filter_view::iterator` operations [ranges.adaptors.filter_view.iterator.ops]

10.8.5.2.1.1 `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`.

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

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

1 *Returns:* `current_`.

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

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

1 *Returns:* `*current_`.

10.8.5.2.1.4 `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;
```

10.8.5.2.1.5 `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;
```

10.8.5.2.1.6 `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)`.

10.8.5.2.2 `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_))
```

10.8.5.3 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);
  };
}}}}
```

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

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

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

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

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

1 *Returns:* `end_`.

10.8.5.3.3 `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)`.

10.8.6 `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.

10.8.7 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 : public 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>>)ViewableRange<O> &&
          Constructible<R, all_view<O>>
    constexpr transform_view(O&& o, F fun);

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

    constexpr R base() const;

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

    constexpr sentinelauto end();
    constexpr const_sentinelauto end() const requires Range<const R> &&
      Invocable<const F&, reference_t<iterator_t<const R>>>;
    constexpr iteratorauto end() requires BoundedRange<R>;
    constexpr const_iteratorauto 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>;
}}}}
```

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

10.8.7.1.1 `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>>)ViewableRange<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)`.

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

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

1 *Returns:* `base_`.

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

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

1 *Effects:* Equivalent to:

```
return __iterator<false>{*this, ranges::begin(base_)};
```

and

```
return __iterator<true>{*this, ranges::begin(base_)};
```

for the first and second overload, respectively.

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

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

1 *Effects:* Equivalent to:

```
return ~~sentinel~~__sentinel<false>{ranges::end(base_)};
```

and

```
return ~~const_sentinel~~__sentinel<true>{ranges::end(base_)};
```

for the first and second overload, respectively.

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

2 *Effects:* Equivalent to:

```
return __iterator<false>{*this, ranges::end(base_)};
```

and

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

for the first and second overload, respectively.

10.8.7.1.5 `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_)`.

10.8.7.2 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>>;
  };
}}}}
```

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

10.8.7.2.1.1 `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_`.

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

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

1 *Returns:* `current_`.

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

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

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

10.8.7.2.1.4 `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;
```

10.8.7.2.1.5 `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;
```

10.8.7.2.1.6 `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;
```

10.8.7.2.1.7 `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]);
```

10.8.7.2.2 `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)`.

10.8.7.2.3 `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_))
```

10.8.7.3 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>>;
  };
}}}}
```

10.8.7.4 `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_`.

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

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

1 *Returns:* `end_`.

10.8.7.6 `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)`.

10.8.7.7 `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_`.

10.8.8 `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.

10.8.9 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 : squaresindices)
  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>
  structclass iota_view : public view_interface<iota_view<I, Bound>> {
  private:
    I value_ {}; // exposition only
    IBound bound_ {}; // exposition only
  public:
    iota_view() = default;
    constexpr explicit iota_view(I value); requires Same<Bound, unreachable>
    constexpr iota_view(I value, Bound bound); // see below

    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`.

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

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

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

1 *Requires:* `Bound{}` is reachable from `value`.

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

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

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

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

5 *Remarks:* This constructor does not contribute a function template to the overload set used when resolving a placeholder for a deduced class type (16.3.1.8).

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

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

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

10.8.9.1.3 `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_}`.

10.8.9.1.4 `iota_view` range size [ranges.adaptors.iota_view.size]

```
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>
```

10.8.9.2 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*]

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

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

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

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

10.8.9.2.1.2 `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*]

10.8.9.2.1.3 `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;
```

10.8.9.2.1.4 `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;
```

10.8.9.2.1.5 `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;
```

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

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

1 *Returns:* `value_ + n`.

10.8.9.2.2 `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)`.

10.8.9.2.3 `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`.

10.8.9.3 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);
  };
}}}}
```

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

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

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

10.8.9.3.2 `iota_view::sentinel` comparisons [ranges.adaptors.iota_view.sentinel.cmp]

```
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)`.

10.8.10 `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.

10.8.11 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~~class take_view : public 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>>)~~ViewableRange<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`.

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

10.8.11.1.1 `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>>)~~ViewableRange<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`.

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

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

1 *Returns:* `base_`.

10.8.11.1.3 `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_);
```

10.8.11.1.4 `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~~`*simple-view*`<R>>{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();
```

10.8.11.1.5 `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_)`.

10.8.11.2 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>>;
  };
}}}}
```

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

10.8.11.2.1.1 `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_`.

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

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

1 *Returns:* `end_`.

10.8.11.2.2 `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)`.

10.8.12 `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.

10.8.13 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 : public 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>>)ViewableRange<O> &&
          Constructible<R, all_view<O>>
    constexpr explicit join_view(O&& o);

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

  constexpr iteratorauto begin();

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

  constexpr sentinelauto end();

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

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

  constexpr const_iteratorauto 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>>;
}}}}
```

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

10.8.13.1.1 `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>>)ViewableRange<O> &&
      Constructible<R, all_view<O>>
constexpr explicit join_view(O&& o);
```

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

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

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

1 *Effects:* Equivalent to:

```
return __iterator<simple-view<R>>{*this, ranges::begin(base_)};
```

and

```
return __iterator<true>{*this, ranges::begin(base_)};
```

for the first and second overloads, respectively. .

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

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

1 *Effects:* Equivalent to:

```
return ~~sentinel~~__sentinel<simple-view<R>>{*this};
```

and

```
return ~~const_sentinel~~__sentinel<true>{*this};
```

for the first and second overload, respectively.

```
constexpr ~~iterator~~auto end() requires ForwardRange<R> &&
  is_reference_v<InnerRng> && ForwardRange<InnerRng> &&
  BoundedRange<R> && BoundedRange<InnerRng>;
constexpr ~~const_iterator~~auto 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 *Effects:* Equivalent to:

```
return __iterator<simple-view<R>>{*this, ranges::end(base_)};
```

and

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

for the first and second overloads, respectively.

10.8.13.2 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_)
```

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

10.8.13.2.1.1 `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_`.

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

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

1 *Returns:* `*inner_`.

10.8.13.2.1.3 `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;
```

10.8.13.2.1.4 `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;
```

10.8.13.2.2 `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)`.

10.8.13.2.3 `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_))
```

10.8.13.3 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);
```

```
  };
}}}}
```

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

10.8.13.3.1.1 `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_`.

10.8.13.3.2 `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)`.

10.8.14 `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.

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

1 The purpose of `empty_view` is to produce an empty 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; }~~is_object_v<T>
  class empty_view : public 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;
  };
}}}}
```

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

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

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

1 *Returns:* `nullptr`.

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

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

1 *Returns:* `nullptr`.

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

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

1 *Returns:* 0.

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

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

1 *Returns:* `nullptr`.

10.8.16 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 : public 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>>;
}}}}
```

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

10.8.16.1.1 `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)...}`.

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

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

1 *Requires:* `bool(value_)`

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

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

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

1 *Requires:* `bool(value_)`

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

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

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

1 *Requires:* `bool(value_)`

2 *Returns:* `1`.

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

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

1 *Requires:* `bool(value_)`

2 *Returns:* `begin()`.

10.8.17 `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.

10.8.18 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>)
  structclass 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>>)ViewableRange<O> &&
          (is_lvalue_reference_v<P> || View<decay_t<P>>)ViewableRange<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>>)ViewableRange<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 iteratorauto begin();
    constexpr iteratorauto begin() requires ForwardRange<Rng>;
    constexpr const_iteratorauto begin() const requires ForwardRange<const Rng>;

    constexpr sentinelauto end()
    constexpr const_sentinelauto end() const requires ForwardRange<const Rng>;

    constexpr iteratorauto end()
      requires ForwardRange<Rng> && BoundedRange<Rng>;
    constexpr const_iteratorauto end() const
      requires ForwardRange<const Rng> && BoundedRange<const Rng>;
  };

  template <InputRange O, ForwardRange P>
    requires (is_lvalue_reference_v<O> || View<decay_t<O>>)ViewableRange<O> &&
      (is_lvalue_reference_v<P> || View<decay_t<P>>)ViewableRange<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>>)ViewableRange<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>>>>;
}}}}
```

10.8.18.1 split_view operations [ranges.adaptors.split_view.ops]

10.8.18.1.1 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>>)~~ViewableRange<O> &&
    ~~(is_lvalue_reference_v<P> || View<decay_t<P>>)~~ViewableRange<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>>)~~ViewableRange<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)}}`.

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

```
constexpr ~~iterator~~auto begin();
```

1 *Effects:* Equivalent to:

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

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

2 *Effects:* Equivalent to:

```
return __outer_iterator<simple-view<R>>{*this, ranges::begin(base_)};
```

and

```
return __outer_iterator<true>{*this, ranges::begin(base_)};
```

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

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

1 *Effects:* Equivalent to:

```
return ~~sentinel~~__outer_sentinel<simple-view<R>>{*this};
```

and

```
return ~~const_sentinel~~__outer_sentinel<true>{*this};
```

for the first and second overloads, respectively.

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

2 *Effects:* Equivalent to:

```
return __outer_iterator<simple-view<R>>{*this, ranges::end(base_)};
```

and

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

10.8.18.2 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`.

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

10.8.18.3.1 `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_`.

10.8.18.3.2 `split_view::__outer_iterator::operator*` [ranges.adaptors.split_view.outer_iterator.star]

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

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

10.8.18.3.3 `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~~subrange{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;
```

10.8.18.3.4 `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);
```

10.8.18.4 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);
  };
}}}}
```

10.8.18.4.1 `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_)`.

10.8.18.4.2 `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);
```

10.8.18.5 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 iteratorauto begin() const;
    constexpr sentinelauto end() const;
  };
}}}}
```

10.8.18.5.1 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`.

10.8.18.5.2 split_view::__outer_iterator::value_type range begin [ranges.adaptors.split_view.outer_iterator.value_type.begin]

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

1 *Effects:* Equivalent to:

```
return ~~iterator~~__inner_iterator<Const>{i_};
```

10.8.18.5.3 split_view::__outer_iterator::value_type range end [ranges.adaptors.split_view.outer_iterator.value_type.end]

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

1 *Effects:* Equivalent to:

```
return ~~sentinel~~__inner_sentinel<Const>{};
```

10.8.18.6 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>>;
  };
}}}}
```

10.8.18.6.1 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`.

10.8.18.6.2 split_view::__inner_iterator::operator* [ranges.adaptors.split_view.inner_iterator.star]

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

1 *Returns:* `*current(i_)`.

10.8.18.6.3 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;
```

10.8.18.6.4 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);
```

10.8.18.6.5 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_)))
```

10.8.18.7 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);
  };
}}}}
```

10.8.18.7.1 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~~subrange{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);
```

10.8.19 `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>>)`~~`ViewableRange<T> &&`
~~`(is_lvalue_reference_v<U> || View<decay_t<U>>)`~~`ViewableRange<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>>)`~~`ViewableRange<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.

10.8.20 `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`~~`subrange{E, E + F}` ~~is~~if `T` is a pointer to an object type, and if `U` is implicitly convertible to `ptrdiff_t`.

(1.2) — Otherwise, ~~`iterator_range`~~`subrange{`~~`make_`~~`counted_iterator(E,` ~~`F`~~`static_cast<difference_type_t<T>>(F)), default_sentinel{}}` if `Iterator<T> && ConvertibleTo<U, difference_type_t<T>>` is satisfied.

(1.3) — Otherwise, `view::counted(E, F)` is ill-formed.

10.8.21 Class template `bounded_view` [ranges.adaptors.bounded_view]

1 The `bounded_view` takes a range which has different types for its iterator and sentinel and turns it into an equivalent range where the iterator and sentinel have the same type.

2 *Remark:* `bounded_view` is useful for calling legacy algorithms that expect a range's iterator and sentinel types to be the same.

3 [*Example:*

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

template <ForwardRange R>
void my_algo(R&& r) {
  auto&& bounded = bounded_view{r};
  auto cnt = count(bounded.begin(), bounded.end());
  // ...
}
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <View Rng>
    requires !BoundedRange<Rng>
  class bounded_view : public view_interface<bounded_view<Rng>> {
  private:
    Rng base_ {}; // exposition only
  public:
    bounded_view() = default;

    explicit constexpr bounded_view(Rng rng);

    template <ViewableRange O>
      requires !BoundedRange<O> && Constructible<Rng, all_view<O>>
    explicit constexpr bounded_view(O&& o);

    constexpr Rng base() const;

    constexpr auto begin();
    constexpr auto begin() const requires Range<const Rng>;

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

    constexpr auto end();
    constexpr auto end() const requires Range<const Rng>;

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

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

```
    template <ViewableRange O>
      requires !BoundedRange<O>
    bounded_view(O&&) -> bounded_view<all_view<O>>;
  }}}}
```

10.8.21.1 bounded_view operations [ranges.adaptors.bounded_view.ops]

10.8.21.1.1 bounded_view constructors [ranges.adaptors.bounded_view.ctor]

```
explicit constexpr bounded_view(Rng base);
```

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

```
template <ViewableRange O>
  requires !BoundedRange<O> && Constructible<Rng, all_view<O>>
explicit constexpr bounded_view(O&& o);
```

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

10.8.21.1.2 bounded_view conversion [ranges.adaptors.bounded_view.conv]

```
constexpr Rng base() const;
```

1 *Returns:* `base_`.

10.8.21.1.3 bounded_view begin [ranges.adaptors.bounded_view.begin]

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

1 *Effects:* Equivalent to:

```
return common_iterator<iterator_t<Rng>, sentinel_t<Rng>>(ranges::begin(base_));
```

and

```
return common_iterator<iterator_t<const Rng>, sentinel_t<const Rng>>(ranges::begin(base_));
```

for the first and second overloads, respectively.

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

2 *Effects:* Equivalent to:

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

10.8.21.1.4 bounded_view end [ranges.adaptors.bounded_view.end]

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

1 *Effects:* Equivalent to:

```
return common_iterator<iterator_t<Rng>, sentinel_t<Rng>>(ranges::end(base_));
```

and

```
return common_iterator<iterator_t<const Rng>, sentinel_t<const Rng>>(ranges::end(base_));
```

for the first and second overloads, respectively.

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

2 *Effects:* Equivalent to:

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

10.8.21.1.5 `bounded_view` size [ranges.adaptors.bounded_view.size]

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

1 *Effects:* Equivalent to: `return ranges::size(base_);`.

10.8.22 `view::bounded` [ranges.adaptors.bounded]

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

(1.1) — If `ViewableRange<U> && BoundedRange<U>` is satisfied, `view::all(E)`.

(1.2) — Otherwise, if `ViewableRange<U>` is satisfied, `bounded_view{E}`.

(1.3) — Otherwise, `view::bounded(E)` is ill-formed.

10.8.23 Class template `reverse_view` [ranges.adaptors.reverse_view]

1 The `reverse_view` takes a bidirectional range and produces another range that iterates the same elements in reverse order.

2 [*Example:*

```
vector<int> is {0,1,2,3,4};
reverse_view rv {is};
for (int i : rv)
  cout << i << ' '; // prints: 4 3 2 1 0
```

— *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <View Rng>
    requires BidirectionalRange<Rng>
  class reverse_view : public view_interface<reverse_view<Rng>> {
  private:
    Rng base_ {}; // exposition only
  public:
    reverse_view() = default;

    explicit constexpr reverse_view(Rng rng);

    template <ViewableRange O>
      requires BidirectionalRange<O> && Constructible<Rng, all_view<O>>
    explicit constexpr reverse_view(O&& o);

    constexpr Rng base() const;

    constexpr auto begin();
```

```
    constexpr auto begin() requires BoundedRange<Rng>;
    constexpr auto begin() const requires BoundedRange<const Rng>;

    constexpr auto end();
    constexpr auto end() const requires BoundedRange<const Rng>;

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

  template <ViewableRange O>
    requires BidirectionalRange<O>
  reverse_view(O&&) -> reverse_view<all_view<O>>;
}}}}
```

10.8.23.1 `reverse_view` operations [ranges.adaptors.reverse_view.ops]

10.8.23.1.1 `reverse_view` constructors [ranges.adaptors.reverse_view.ctor]

```
explicit constexpr reverse_view(Rng base);
```

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

```
template <ViewableRange O>
  requires !BoundedRange<O> && Constructible<Rng, all_view<O>>
explicit constexpr reverse_view(O&& o);
```

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

10.8.23.1.2 `reverse_view` conversion [ranges.adaptors.reverse_view.conv]

```
constexpr Rng base() const;
```

1 *Returns:* `base_`.

10.8.23.1.3 `reverse_view` begin [ranges.adaptors.reverse_view.begin]

```
constexpr auto begin();
```

1 *Effects:* Equivalent to:

```
return reverse_iterator{ranges::next(ranges::begin(base_), ranges::end(base_))};
```

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

```
constexpr auto begin() requires BoundedRange<Rng>;
constexpr auto begin() const requires BoundedRange<const Rng>;
```

3 *Effects:* Equivalent to:

```
return reverse_iterator{ranges::end(base_)};
```

10.8.23.1.4 `reverse_view` end [ranges.adaptors.reverse_view.end]

```
constexpr auto end() requires BoundedRange<Rng>;
constexpr auto end() const requires BoundedRange<const Rng>;
```

1 *Effects:* Equivalent to:

```
return reverse_iterator{ranges::begin(base_)};
```

10.8.23.1.5 `reverse_view` size [ranges.adaptors.reverse_view.size]

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

1 *Effects:* Equivalent to:

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

10.8.24 `view::reverse` [ranges.adaptors.reverse]

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

(1.1) — If `ViewableRange<U> && BidirectionalRange<U>` is satisfied, `reverse_view{E}`.

(1.2) — Otherwise, `view::reverse(E)` 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.

Index

Index of library names