Document Number: P1035R5
Date: 2019-03-05
Audience: Library Working Group
Authors: Christopher Di Bella
Casey Carter
Corentin Jabot
Reply to: Christopher Di Bella
cjdb.ns@gmail.com

Input Range Adaptors

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

Contents

1 Scope [intro.scope]

1 This document proposes to merge the range adaptors described below with the C++20 Working Draft.

1.1 Revision History [intro.history]

1.1.1 Revision 5 [intro.history.r5]

— Removed `zip_view`-related sections, as requested by LEWG.

— Removed *constructible-from-range* constructor as per LEWG discussion.

— Weakened the `Semiregular<Val>` requirement to `Movable<Val> && DefaultConstructor<Val>` for `basic_istream_view`.

— (Editorial) Migrated from Bikeshed HTML to LaTeX.

— Adds editorial changes such as ~~`iter_value_t<iterator_t<R>>`~~`range_value_t<R>` for review by LWG to simplify text in the International Standard.

1.1.2 Revision 4 [intro.history.r4]

— Proposes that `iterator_t` and `sentinel_t` require `Range` in their interface.

— Adjusts associated types for ranges so that they don't explicitly require `Range` (this is deferred to `iterator_t`).

1.1.3 Revision 3 [intro.history.r3]

— Adds polls from San Diego meeting.

— Removed `range_size_t` and `range_common_iterator_t` from the associated types.

— Added justification for why `is_object_v` is necessary for `take_while_view`.

— Replaced contract-specified pre-conditions with text-specified pre-conditions.

— Removed concept `StreamInsertable`, as it is not relevant to the contents of this paper.

— Replaced concept `StreamExtractable` with exposition-only concept *stream-extractable*.

  — This was done, in part, to balance the fact that a concept would exist for `operator>>` but not `operator<<`.

— Replaced pros and cons of `__tuple_hack` with const-qualified overloads for `std::tuple` and necessary `common_type` and `basic_common_reference` specialisations.

1.1.4 Revision 2 [intro.history.r2]

— Expanded acknowledgements and co-authors.

— Removed `zip_with_view`.

— Added `zip_view`.

— Added `keys` and `values`.

— Added content for associated types for ranges.

1.1.5 Revision 1 [intro.history.r1]

— Revised `istream_range`.

— Renamed to `basic_istream_view`.

— Introduced some relevant concepts.

— Introduced `drop_view`, `take_while_view`, `drop_while_view`.

— Teased `zip_with_view`.

— Teased associated types for ranges.

1.1.6 Revision 0 [intro.history.r1]

— Initial proposal.

2 General Principles [intro]

> "Law III: To every action there is always opposed an equal reaction: or the mutual actions of two bodies upon each other are always equal, and directed to contrary parts."
>
> —*Isaac Newton's Third Law of Motion*

2.1 Goals [intro.goals]

1 The primary goal of this paper is to extend the number of range adaptors present in C++20.

2.2 Rationale [intro.rationale]

1 P0789 – and by extension, P0896 – merged twelve range adaptors into the C++20 Working Draft. Due to the finite amount of time that the authors of P0896 have, this is only a glimpse of the range adaptors that can be added to C++for declarative programming. P1035 adds another four complimentary range adaptors to 'complete' the C++20 suite of range adaptors.

2.3 Style of presentation [intro.style]

1 The remainder of this document is a technical specification in the form of editorial instructions directing that changes be made to the text of the C++ working draft. The formatting of the text suggests the origin of each portion of the wording.

Existing wording from the C++ working draft - included to provide context - is presented without decoration.

Entire clauses / subclauses / paragraphs incorporated from the Ranges TS are presented in a distinct teal color.

In-line additions of wording from P1035 to the C++ working draft are presented in teal with underline.

~~In-line bits of wording that P1035 strikes from the C++ working draft are presented in red with strike-through.~~

Wording to be added which is original to this document appears in gold with underline.

~~Wording which this document strikes is presented in magenta with strikethrough. (Hopefully context makes it clear whether the wording is currently in the C++ working draft, or wording that is not being added from the Ranges TS.)~~

Ideally, these formatting conventions make it clear which wording comes from which document in this three-way merge.

19 General utilities library [utilities]

[...]

19.10 Memory [memory]

19.10.2 Header <memory> synopsis [memory.syn]

[...]

```
namespace std {
  // ...
  namespace ranges {
    // ...
    template<NoThrowForwardRange R>
      requires DefaultConstructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>>
        safe_iterator_t<R> uninitialized_default_construct(R&& r);
  }
  // ...
  namespace ranges {
    // ...
    template<NoThrowForwardRange R>
      requires DefaultConstructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>>
        safe_iterator_t<R> uninitialized_value_construct(R&& r);
  }
  // ...
  namespace ranges {
    // ...
    template<InputRange IR, NoThrowForwardRange OR>
    ~~requires Constructible<iter_value_t<iterator_t<OR>>, iter_reference_t<iterator_t<IR>>>~~
    requires Constructible<range_value_t<OR>, range_reference_t<IR>>
      uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_copy(IR&& input_range, OR&& output_range);
  }
  namespace ranges {
    template<InputRange IR, no-throw-forward-range OR>
    requires Constructible<~~iter_value_t<iterator_t<OR>>~~range_value_t<OR>,
                           ~~iter_rvalue_reference_t<iterator_t<IR>>~~>range_rvalue_reference_t<IR>
      uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_move(IR&& input_range, OR&& output_range);
    // ...
  }
  // ...
  namespace ranges {
    // ...
    template<NoThrowForwardRange R, class T>
      requires Constructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>, const T&>
        safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);
  }
  // ...
  namespace ranges {
    // ...
    template<NoThrowInputRange R>
      requires Destructible<~~iter_value_t<iterator_t<R>~~range_value_t<R>>
        safe_iterator_t<R> destroy(R&& r) noexcept;
  }
  // ...
}
```

[...]

19.10.11 Specialized algorithms [specialized.algorithms]

[...]

19.10.11.1 uninitialized_default_construct [uninitialized.construct.default]

[...]

```
namespace ranges {
  // ...
  template<NoThrowForwardRange R>
      requires DefaultConstructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>>
    safe_iterator_t<R> uninitialized_default_construct(R&& r);
}
```

[...]

19.10.11.2 uninitialized_value_construct [uninitialized.construct.value]

[...]

```
namespace ranges {
  // ...
  template<NoThrowForwardRange R>
      requires DefaultConstructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>>
    safe_iterator_t<R> uninitialized_value_construct(R&& r);
}
```

[...]

19.10.11.3 uninitialized_copy [uninitialized.copy]

[...]

```
  namespace ranges {
    template<InputRange IR, no-throw-forward-range OR>
    requires Constructible<~~iter_value_t<iterator_t<OR>>~~range_value_t<OR>,
                           ~~iter_rvalue_reference_t<iterator_t<IR>>>~~range_rvalue_reference_t<IR>
      uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_move(IR&& input_range, OR&& output_range);
    // ...
  }
```

[...]

19.10.11.4 uninitialized_move [uninitialized.move]

[...]

```
  namespace ranges {
    // ...
    template<NoThrowForwardRange R, class T>
      requires Constructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>, const T&>
        safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);
  }
```

[...]

19.10.11.5 uninitialized_fill [uninitialized.fill]

[...]

```
namespace ranges {
  // ...
  template<NoThrowForwardRange R, class T>
      requires Constructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>, const T&>
    safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);
}
```

[...]

19.10.11.6 `destroy` [specialized.destroy]

[...]

```
namespace ranges {
  // ...
  template<NoThrowInputRange R>
      requires Destructible<~~iter_value_t<iterator_t<R>>~~range_value_t<R>>
    safe_iterator_t<R> destroy(R&& r) noexcept;
}
```

22 Iterators library [iterators]

22.1 Header <iterator> synopsis [iterator.synopsis]

```
namespace std {
  // ...
  namespace ranges {
    // ...
    // (22.1.0.1), ranges::distance
    template<Iterator I, Sentinel<I> S>
    constexpr iter_difference_t<I> distance(I first, S last);
    template<Range R>
    constexpr ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> distance(R&& r);
    // ...
  }
  // ...
}
```

[...]

22.1.0.1 ranges::distance [range.iterator.operations.distance]

[...]

```
template<Range R>
  constexpr ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> distance(R&& r);
```

[...]

23 Ranges library [range]

23.1 Header <ranges> synopsis [ranges.syn]

[...]

```
#include <initializer_list>
#include <iterator>
// ...
namespace std::ranges {
  // ??, Range
  template<class T>
    using iterator_t = decltype(ranges::begin(declval<T&>()));

  template<class T>
    using sentinel_t = decltype(ranges::end(declval<T&>()));

  template<fowarding-range R>
    using safe_iterator_t = iterator_t<R>;

  template<class T>
    concept Range = see below;

  template<Range R>
    using iterator_t = decltype(ranges::begin(declval<T&>()));

  template<Range T>
    using sentinel_t = decltype(ranges::end(declval<T&>()));

  template<fowarding-range R>
    using safe_iterator_t = iterator_t<R>;

  template<class R>
    using range_difference_t = iter_difference_t<iterator_t<R>>;

  template<class R>
    using range_value_t = iter_value_t<iterator_t<R>>;

  template<class R>
    using range_reference_t = iter_reference_t<iterator_t<R>>;

  template<class R>
    using range_rvalue_reference_t = iter_rvalue_reference_t<iterator_t<R>>;

  // ??, SizedRange
  // ...

  // 23.7.5, transform view
  template<InputRange V, CopyConstructible F>
    requires View<V> && is_object_v<F> &&
        RegularInvocable<F&, iter_reference_t<iterator_t<V>>range_reference_t<V>>
  class transform_view;

  // 23.7.6, take view
  // ...

  // 23.7.10, join view
  // ...

  // 23.7.12, split view
  // ...
```

```
// 23.7.13, counted view
// ...

// 23.7.14, common view
// ...

// 23.7.15, reverse view
// ...

// 23.7.7, take_while view
template<View R, class Pred>
  requires InputRange<R> && is_object_v<Pred> &&
    IndirectUnaryPredicate<const Pred, iterator_t<R>>
class take_while_view;

namespace view { inline constexpr unspecified take_while = unspecified; }

// 23.7.8, drop view
template<View R>
  class drop_view;

namespace view { inline constexpr unspecified drop = unspecified; }

// 23.7.9, drop_while view
template<View R, class Pred>
  requires InputRange<R> && is_object_v<Pred> &&
    IndirectUnaryPredicate<const Pred, iterator_t<R>>
class drop_while_view;

namespace view { inline constexpr unspecified drop_while = unspecified; }

// 23.7.10, join view
template<InputRange V>
  requires View<V> && InputRange<iter_reference_t<iterator_t<V>>range_reference_t<V>> &&
      (is_reference_v<iter_reference_t<iterator_t<V>>range_reference_t<V>> ||
       View<iter_value_t<iterator_t<V>>range_value_t<V>>)
class join_view;

// 23.7.12, split view
// ...

// 23.7.13, counted view
// ...

// 23.7.14, common view
// ...

// 23.7.15, reverse view
// ...

// 23.7.16, istream view
template<class T, class CharT, class Traits>
  concept stream-extractable = see below; // exposition only

template<Movable Val, class CharT, class Traits = char_traits<CharT>>
  requires DefaultConstructible<Val> && stream-extractable<Val, CharT, Traits>
class basic_istream_view;

template<Movable T, class CharT, class Traits>
requires DefaultConstructible<Val> && stream-extractable<Val, CharT, Traits>
basic_istream_view<T, CharT, Traits> istream_view(basic_istream<CharT, Traits>& s);
```

```
  // 23.7.17, elements view
  template<class T, size_t N>
    concept tuple-like = see below; // exposition only

  template<InputRange R, size_t N>
    requires View<R> && tuple-like<range_value_t<R>, N> &&
      tuple-like<remove_reference_t<range_reference_t<R>>, N>
  class elements_view;

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

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

23.5 Range requirements [range.req]

[...]

23.5.4 Views [range.view]

[...]

```
template<class T>
  inline constexpr bool enable_view = see below;

template<class T>
  concept View =
    Range<T> && Semiregular<T> && enable_view<T>;
```

3 Since the difference between `Range` and `View` is largely semantic, the two are differentiated with the help of `enable_view`.

4 For a type `T`, the default value of `enable_view<T>` is:

(4.1) — If `DerivedFrom<T, view_base>` is `true`, `true`.

(4.2) — Otherwise, if `T` is a specialization of class template `initializer_list` ([support.initlist]), `set` ([set]), `multiset` ([multiset]), `unordered_set` ([unord.set]), `unordered_multiset` ([unord.multiset]), or `match_results` ([re.results]), `false`.

(4.3) — Otherwise, if both `T` and `const T` model `Range` and ~~`iter_reference_t<iterator_t<T>>`~~`range_reference_t<T>` is not the same type as ~~`iter_reference_t<iterator_t<const T>>`~~`range_reference_t<T>`, `false`. [*Note*: Deep `const`-ness implies element ownership, whereas shallow `const`-ness implies reference semantics. — *end note*]

(4.4) — Otherwise, `true`.

5 Pursuant to [namespace.std], users may specialize `enable_view` to `true` for types which model `View`, and `false` for types which do not.

[...]

23.5.5 Common range refinements [range.refinements]

[...]

```
template<class T>
  concept ContiguousRange =
    RandomAccessRange<T> && ContiguousIterator<iterator_t<T>> &&
    requires(T& t) {
      ranges::data(t);
      requires Same<decltype(ranges::data(t)), add_pointer_t<~~iter_reference_t<iterator_t<T>>~~range_reference_t<T>>>
```

```
  };
```

[...]

23.6 Range utilities [range.utility]

23.6.1 Helper concepts [range.utility.helpers]

[...]

23.6.2 View interface [view.interface]

[...]

```
namespace std::ranges {
  // ...
  template<class D>
    requires is_class_v<D> && Same<D, remove_cv_t<D>>
  class view_interface : public view_base {
  private:
    // ...
    template<RandomAccessRange R = D>
      constexpr decltype(auto) operator[](~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> n) {
        return ranges::begin(derived())[n];
      }
    template<RandomAccessRange R = const D>
      constexpr decltype(auto) operator[](~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> n) const {
        return ranges::begin(derived())[n];
      }
  };
}
```

23.6.3 Sub-ranges [range.subrange]

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

```
namespace std::ranges {
  // ...
  template<forwarding-range R>
    subrange(R&&, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R>) ->
      subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

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

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

23.7 Range adaptors [range.adaptors]

23.7.4 Filter view [range.filter]

23.7.4.3 Class template `filter_view::iterator` [range.filter.iterator]

```
namespace std::ranges {
  template<class V, class Pred>
  class filter_view<V, Pred>::iterator {
    // ...
  public:
    using iterator_concept  = see below;
    using iterator_category = see below;
    using value_type        = ~~iter_value_t<iterator_t<V>>~~range_value_t<V>;
    using difference_type   = ~~iter_difference_t<iterator_t<V>>~~range_difference_t<V>;
```

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

    constexpr iterator_t<V> base() const;
    constexpr ~~iter_reference_t<iterator_t<V>>~~range_reference_t<V> operator*() const;

    // ...

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

[...]

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

Effects: Equivalent to: `return *current_;`

[...]

```
friend constexpr ~~iter_rvalue_reference_t<iterator_t<V>>~~range_rvalue_reference_t<V> iter_move(const iterator& i)
  noexcept(noexcept(ranges::iter_move(i.current_)));
```

Effects: Equivalent to: `return ranges::iter_move(i.current_);`

[...]

23.7.5 Transform view [range.transform]

23.7.5.1 Overview [range.transform.overview]

[...]

23.7.5.2 Class template `transform_view` [range.transform.view]

```
namespace std::ranges {
  template<InputRange V, CopyConstructible F>
    requires View<V> && is_object_v<F> &&
             RegularInvocable<F&, ~~iter_reference_t<iterator_t<V>>~~range_reference_t<V>>
  class transform_view : public view_interface<transform_view<V, F>> {
  private:
    // ...
  public:
    // ...

    constexpr iterator<false> begin();
    constexpr iterator<true> begin() const
      requires Range<const V> &&
               RegularInvocable<const F&, ~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const
V>>;

    constexpr sentinel<false> end();
    constexpr iterator<false> end() requires CommonRange<V>;
    constexpr sentinel<true> end() const
      requires Range<const V> &&
               RegularInvocable<const F&, ~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const
V>>;
    constexpr iterator<true> end() const
      requires CommonRange<const V> &&
               RegularInvocable<const F&, ~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const
V>>;
```

```
    // ...
  };
}
```

[...]

```
constexpr iterator<true> begin() const
  requires Range<const V> &&
           RegularInvocable<const F&, ~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>>;
```

Effects: Equivalent to:

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

[...]

```
constexpr sentinel<true> end() const
  requires Range<const V> &&
           RegularInvocable<const F&, ~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>>;
```

Effects: Equivalent to:

```
return sentinel<true>{ranges::end(base_)};;
```

```
constexpr iterator<true> end() const
  requires CommonRange<const V> &&
  RegularInvocable<const F&, ~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>>;
```

Effects: Equivalent to:

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

[...]

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

```
namespace std::ranges {
  template<class V, class F>
  template<bool Const>
  class transform_view<V, F>::iterator {
  private:
    // ...
  public:
    using iterator_concept = see below;
    using iterator_category = see below;
    using value_type =
      remove_cvref_t<invoke_result_t<F&, ~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>>;
    using difference_type = ~~iter_difference_t<iterator_t<Base>>~~range_difference_t<Base>;
    // ...
  };
}
```

23.7.5.4 Class template `transform_view::sentinel` [range.transform.sentinel]

```
namespace std::ranges {
  template<class V, class F>
  template<bool Const>
  class transform_view<V, F>::sentinel<Const> {
  private:
    // ...
  public:
    // ...
    friend constexpr ~~iter_difference_t<iterator_t<Base>>~~range_difference_t<Base>
      operator-(const iterator<Const>& x, const sentinel& y)
        requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
    friend constexpr ~~iter_difference_t<iterator_t<Base>>~~range_difference_t<Base>
      operator-(const sentinel& y, const iterator<Const>& x)
        requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
  };
}
```

[...]

```
friend constexpr ~~iter_difference_t<iterator_t<Base>>~~range_difference_t<Base>
  operator-(const iterator<Const>& x, const sentinel& y)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

8 *Effects:* Equivalent to: `return x.current_ - y.end_;`

```
friend constexpr ~~iter_difference_t<iterator_t<Base>>~~range_difference_t<Base>
  operator-(const sentinel& y, const iterator<Const>& x)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
```

9 *Effects:* Equivalent to: `return x.end_ - y.current_;`

23.7.6 Take view [range.take]

23.7.6.1 Overview [range.take.overview]

[...]

23.7.6.2 Class template `take_view` [range.take.view]

```
namespace std::ranges {
  template<View V>
  class take_view : public view_interface<take_view<V>> {
  private:
    V base_ = V();                          // exposition only
    ~~iter_difference_t<iterator_t<V>>~~range_difference_t<V> count_ = 0; // exposition only
    template<bool> struct sentinel;         // exposition only
  public:
    take_view() = default;
    constexpr take_view(V base, ~~iter_difference_t<iterator_t<V>>~~range_difference_t<V> count);
    template<ViewableRange R>
      requires Constructible<V, all_view<R>>
    constexpr take_view(R&& r, ~~iter_difference_t<iterator_t<V>>~~range_difference_t<V> count);
    // ...
  };

  template<Range R>
  take_view(R&&, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R>)
    -> take_view<all_view<R>>;
}
```

```
constexpr take_view(V base, ~~iter_difference_t<iterator_t<V>>~~range_difference_t<V> count);
```

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

```
template<ViewableRange R>
  requires Constructible<V, all_view<R>>
constexpr take_view(R&& r, ~~iter_difference_t<iterator_t<V>>~~range_difference_t<V> count);
```

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

[...]

23.7.7 Join view [range.join]

[...]

23.7.8 Split view [range.split]

[...]

23.7.9 Counted view [range.counted]

[...]

23.7.10 Common view [range.common]

[...]

23.7.11 Reverse view [range.reverse]

[...]

23.7.7 Take while view [range.take_while]

23.7.7.1 Overview [range.take_while.overview]

1 `take_while_view` produces a `View` of the first *N* elements that satisfy the predicate `Pred` from another `View`, or all the elements if the adapted `View` contains no elements that do not satisfy `Pred`.

2 [*Example*:

```
auto ints = iota_view(0);
auto small = [](const auto x) noexcept { return x < 5; };
auto small_ints = take_while_view{ints, small};
for (const auto i : small_ints) {
  cout << i << ' '; // prints 0 1 2 3 4
}
```

— *end example*]

3 [*Note*: `take_while_view` consumes the element that it reads. Users should be aware that this makes `take_while_view` inappropriate for input iterators in contexts where the iterator's value is relevant *after* the range adaptor is used.

[*Example*:

```
auto input = istringstream{"0 1 2 3 4 5 6 7 8 9"};
auto small = [](const auto x) noexcept { return x < 5; };
auto small_ints = istream_view<int>(input)
                | view::take_while(small);
for (const auto i : small_ints) {
   cout << i << ' ' // prints 0 1 2 3 4
}
auto i = 0;
input >> i;
cout << i; // prints 6
```

— *end example*] — *end note*]

23.7.7.2 Class template `take_while_view` [range.take_while.view]

```
namespace std::ranges {
  template<View R, class Pred>
  requires InputRange<R> && is_object_v<Pred> &&
    IndirectUnaryPredicate<const Pred, iterator_t<R>>
  class take_while_view : public view_interface<take_while_view<R, Pred>> {
     template<bool> class sentinel; // exposition only

     R base_;               // exposition only
     semiregular<Pred> pred_; // exposition only
  public:
     take_while_view() = default;
     constexpr take_while_view(R base, Pred pred);

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

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

     constexpr auto end() requires (!simple-view<R>);
     constexpr auto end() const requires Range<const R>;
  };

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

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

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

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

2 *Effects:* Equivalent to: `return base_;`

```
constexpr const Pred& pred() const;
```

3 *Effects:* Equivalent to: `return pred_;`

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

4 *Effects:* Equivalent to: `return begin(base_);`

```
constexpr auto end() requires (!simple-view<R>);
constexpr auto end() const requires Range<const R>;
```

5 *Effects:* Equivalent to: `return sentinel<is_const_v<decltype(*this)>>(addressof(pred()));`

23.7.7.3 Class template `take_while::sentinel` [range.take_while.sentinel]

```
namespace std::ranges {
  template<class V>
  template<bool Const>
  class take_while_view<V>::sentinel {
    using base_t = conditional_t<Const, const V, V>;    // exposition only

    sentinel_t<base_t> end_{}; // exposition only
    const Pred* pred_{};       // exposition only
  public:
    sentinel() = default;
    constexpr explicit sentinel(sentinel_t<base_t> end, const Pred* pred);
    constexpr sentinel(sentinel<!Const> s)
      requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<base_t>>;

    constexpr sentinel_t<base_t> base() const { return end_; }

    friend constexpr bool operator==(const sentinel& x, const iterator_t<base_t>& y);
    friend constexpr bool operator==(const iterator_t<base_t>& x, const sentinel& y);
    friend constexpr bool operator!=(const sentinel& x, const iterator_t<base_t>& y);
    friend constexpr bool operator!=(const iterator_t<base_t>& x, const sentinel& y);
  };
}
```

```
constexpr explicit sentinel(sentinel_t<base> end, const Pred* pred);
```

1 *Effects:* Initializes `end_` with `end` and `pred_` with `pred`.

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

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

```
friend constexpr bool operator==(const sentinel& x, const iterator_t<base>& y);
friend constexpr bool operator==(const iterator_t<base>& y, const sentinel& x);
```

3 *Effects:* Equivalent to: `return x.end_ != y && !(*x.pred_)(*y);`

```
friend constexpr bool operator!=(const sentinel& x, const iterator_t<base>& y);
friend constexpr bool operator!=(const iterator_t<base>& y, const sentinel& x);
```

4 *Effects:* Equivalent to: `return !(x == y);`

23.7.7.4 `view::take_while` [range.take_while.adaptor]

1 The name `view::take_while` denotes a range adaptor object (**??**). For some subexpressions `E` and `F`, the expression `view::take_while(E, F)` is expression-equivalent to `take_while_view{E, F}`.

23.7.8 Drop view [range.drop]

23.7.8.1 Overview [range.drop.overview]

1 `drop_view` produces a `View` excluding the first *N* elements from another `View`, or an empty range if the adapted `View` contains *N* or fewer elements.

2 [*Example*:

```
auto ints = view::iota(0) | view::take(10);
auto latter_half = drop_view{ints, 5};
for (auto i : latter_half) {
  cout << i << ' '; // prints 5 6 7 8 9
}
```

— *end example*]

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

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

    constexpr R base() const;

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

    constexpr auto end()
      requires (!(simple-view<R> && RandomAccessRange<R>));
    constexpr auto end()
      requires Range<const R> && RandomAccessRange<const R>;

    constexpr auto size()
      requires (!(simple-view<R> && RandomAccessRange<R>));
    constexpr auto size()
      requires Range<const R> && RandomAccessRange<const R>;
  private:
    R base_;  // exposition only
    D count_; // exposition only
  };

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

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

1 *Effects:* Initializes `base_` with `base` and `count_` with `count`.

2 *Expects:* `0 < count`.

`constexpr R base() const;`

3 *Effects:* Equivalent to: `return base_;`

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

4 *Effects:* Equivalent to: `return next(begin(base_), count_, end(base_));`

5 *Remarks:* In order to provide the amortized constant-time complexity requirement by the `Range` concept, the first overload caches the result within the `drop_view` for use on subsequent calls. Without this, applying a `reverse_view` over a `drop_while` would have quadratic iteration complexity.

```
constexpr auto end()
  requires (!(simple-view<R> && RandomAccessRange<R>));
constexpr auto end()
  requires Range<const R> && RandomAccessRange<const R>;
```

6 *Effects:* Equivalent to: `return end(base_);`

```
constexpr auto size()
  requires (!(simple-view<R> && RandomAccessRange<R>));
constexpr auto size()
  requires Range<const R> && RandomAccessRange<const R>;
```

7 *Effects:* Equivalent to:

```
const auto s = size(base_);
const auto c = static_cast<decltype(s)>(count_);
return s < c ? 0 : s - c;
```

23.7.8.3 `view::drop` [range.drop.adaptor]

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

23.7.9 Drop while view [range.drop_while]

23.7.9.1 Overview [range.drop_while.overview]

1 `drop_while_view` produces a `View` of the first N elements that satisfy the predicate `Pred` from another `View`, or an empty range if no elements in the adapted `View` satisfy `Pred`.

2 [*Example*:

```
constexpr auto source = "  \t   \t   \t   hello there";
auto is_space = [](const auto x) { return x == ' ' || x == '\t'; };
auto skip_ws = drop_view{source, is_space};
for (auto c : skip_ws) {
  cout << c; // prints hellothere
}
```

— *end example*]

23.7.9.2 Class template `drop_while` [range.drop_while.view]

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

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

    constexpr auto begin();
    constexpr auto end();
  private:
    R base_;                    // exposition only
    semiregular<Pred> pred_;    // exposition only
  };

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

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

1 *Effects:* Initializes `base_` with `base` and initializes `pred_` with `pred`.

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

2 *Effects:* Equivalent to: `return base_;`

```
constexpr const Pred& pred() const
```

3 *Effects:* Equivalent to: `return pred_;`

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

4 *Effects:* Equivalent to: `return find_if_not(base_, std::ref(pred_));`

5 *Remarks:* In order to provide the amortized constant-time complexity required by the `Range` concept, the first call caches the result within the `drop_while_view` for use on subsequent calls. Without this, applying a `reverse_view` over a `drop_while_view` would have quadratic iteration complexity.

```
constexpr auto end();
```

6 *Effects:* Equivalent to: `return end(base_);`

23.7.9.3 `view::drop_while` [range.drop_while.adaptor]

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

23.7.10 Join view [range.join]

[Editor's note: The contents of 23.7.10 has been *moved*. The text is not coloured teal to help the snippets that have *changed* stand out from the sections that are copied verbatim.]

23.7.10.1 Overview [range.join.overview]

[...]

23.7.10.2 Class template `join_view` [range.join.view]

```
namespace std::ranges {
  template<InputRange V>
    requires View<V> && InputRange<~~iter_reference_t<iterator_t<V>>~~range_reference_t<V>> &&
             (is_reference_v<~~iter_reference_t<iterator_t<V>>~~range_reference_t<V>> ||
             View<~~iter_value_t<iterator_t<V>>~~range_value_t<V>>)
  class join_view : public view_interface<join_view<V>> {
  private:
    using InnerRng = // exposition only
      ~~iter_reference_t<iterator_t<V>>~~range_reference_t<V>;
    // ...
  public:
    // ...
    constexpr auto begin() const
    requires InputRange<const V> &&
             is_reference_v<~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>> {
      return iterator<true>{*this, ranges::begin(base_)};
    }
    // ...
    constexpr auto end() const
    requires InputRange<const V> &&
             is_reference_v<~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>> {
      if constexpr (ForwardRange<const V> &&
                    is_reference_v<~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>> &&
                    ForwardRange<~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>> &&
                    CommonRange<const V> &&
                    CommonRange<~~iter_reference_t<iterator_t<const V>>~~range_reference_t<const V>>)
        return iterator<true>{*this, ranges::end(base_)};
      else
        return sentinel<true>{*this};
    }
  };
```

```
    template<class R>
      explicit join_view(R&&) -> join_view<all_view<R>>;
  }
```

[...]

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

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

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

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

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

      iterator() = default;
      constexpr iterator(Parent& parent, iterator_t<V> outer);
      constexpr iterator(iterator<!Const> i)
        requires Const &&
                 ConvertibleTo<iterator_t<V>, iterator_t<Base>> &&
                 ConvertibleTo<iterator_t<InnerRng>,
                               iterator_t<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>>;
      // ...
      constexpr iterator& operator++();
      constexpr void operator++(int);
      constexpr iterator operator++(int)
        requires ref_is_glvalue && ForwardRange<Base> &&
                 ForwardRange<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;

      constexpr iterator& operator--()
        requires ref_is_glvalue && BidirectionalRange<Base> &&
          BidirectionalRange<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;

      constexpr iterator operator--(int)
        requires ref_is_glvalue && BidirectionalRange<Base> &&
          BidirectionalRange<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;

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

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

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

```
    }

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

2 `iterator::iterator_concept` is defined as follows:

(2.1) — If `ref_is_glvalue` is `true`,

(2.1.1) — If `Base` and ~~`iter_reference_t<iterator_t<Base>>`~~`range_reference_t<Base>` each model `BidirectionalRange`, then `iterator_concept` denotes `bidirectional_iterator_tag`.

(2.1.2) — Otherwise, if `Base` and ~~`iter_reference_t<iterator_t<Base>>`~~`range_reference_t<Base>` each model ForwardRange, then `iterator_concept` denotes `forward_iterator_tag`.

(2.2) — Otherwise, `iterator_concept` denotes `input_iterator_tag`.

3 `iterator::iterator_category` is defined as follows:

(3.1) — Let *OUTERC* denote `iterator_traits<iterator_t<Base>>::iterator_category`, and let *INNERC* denote `iterator_traits<iterator_t<`~~`iter_reference_t<iterator_t<Base>>`~~`range_reference_t<Ba` `category`.

(3.2) — If `ref_is_glvalue` is `true`,

(3.2.1) — If *OUTERC* and *INNERC* each model `DerivedFrom<bidirectional_iterator_tag>`, `iterator_category` denotes `bidirectional_iterator_tag`.

(3.2.2) — Otherwise, if *OUTERC* and *INNERC* each model `DerivedFrom<forward_iterator_tag>`, `iterator_category` denotes `forward_iterator_tag`.

(3.3) — Otherwise, `iterator_category` denotes input_iterator_tag.

4 `iterator::difference_type` denotes the type:

```
common_type_t<
  iter_difference_t<iterator_t<Base>>range_difference_t<Base>>,
  iter_difference_t<iterator_t<iter_reference_t<iterator_t<Base>>>>
  range_difference_t<range_reference_t<Base>>>
```

5 `join_view` iterators use the `satisfy` function to skip over empty inner ranges.

```
constexpr void satisfy(); // exposition only
```

6 *Effects:* Equivalent to:

```
auto update_inner = [this](iter_reference_t<iterator_t<Base>>range_reference_t<Base> x) -> decltype(auto)
  if constexpr (ref_is_glvalue) // x is a reference
    return (x); // (x) is an lvalue
  else
    return (parent_->inner_ = view::all(x));
};

for (; outer_ != ranges::end(parent_->base_); ++outer_) {
  auto& inner = update_inner(*outer_);
  inner_ = ranges::begin(inner);
  if (inner_ != ranges::end(inner))
    return;
}

if constexpr (ref_is_glvalue)
  inner_ = iterator_t<iter_reference_t<iterator_t<Base>>range_reference_t<Base>>();
```

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

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

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

```
                    iterator_t<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;
```

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

[...]

```
constexpr iterator operator++(int)
  requires ref_is_glvalue && ForwardRange<Base> &&
           ForwardRange<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;
```

13 *Effects:* Equivalent to:

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

```
constexpr iterator& operator--()
  requires ref_is_glvalue && BidirectionalRange<Base> &&
           BidirectionalRange<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;
```

14 *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 ref_is_glvalue && BidirectionalRange<Base> &&
           BidirectionalRange<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>;
```

15 *Effects:* Equivalent to:

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

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

16 *Effects:* Equivalent to: `return x.outer_ == y.outer_ && x.inner_ == y.inner_;`

```
friend constexpr bool operator!=(const iterator& x, const iterator& y)
  requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&
           EqualityComparable<iterator_t<~~iter_reference_t<iterator_t<Base>>~~range_reference_t<Base>>>;
```

17 *Effects:* Equivalent to: `return !(x == y);`

[...]

23.7.12 Split view [range.split]

[Editor's note: The contents of 23.7.12 has been *moved*. The text is not coloured teal to help the snippets that have *changed* stand out from the sections that are copied verbatim.]

23.7.12.1 Overview [range.split.overview]

[...]

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

```
namepsace std::ranges {
  // ...

  template<InputRange V, ForwardRange Pattern>
    requires View<V> && View<Pattern> &&
             IndirectlyComparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
             (ForwardRange<V> || tiny-range<Pattern>)
  class split_view : public view_interface<split_view<V, Pattern>> {
```

```
    private:
      // ...
    public:
      // ...

      template<InputRange R>
        requires Constructible<V, all_view<R>> &&
                 Constructible<Pattern, single_view<iter_value_t<iterator_t<R>>range_value_t<R>>>
      constexpr split_view(R&& r, iter_value_t<iterator_t<R>>range_value_t<R> e);

      // ...
    };

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

    template<InputRange R>
    split_view(R&&, iter_value_t<iterator_t<R>>range_value_t<R>)
      -> split_view<all_view<R>, single_view<iter_value_t<iterator_t<R>>range_value_t<R>>>;
  }
```

[...]

```
template<InputRange R>
  requires Constructible<V, all_view<R>> &&
           Constructible<Pattern, single_view<iter_value_t<iterator_t<R>>range_value_t<R>>>
  constexpr split_view(R&& r, iter_value_t<iterator_t<R>>range_value_t<R> e);
```

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

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

```
  namespace std::ranges {
    template<class V, class Pattern>
    template<bool Const>
    struct split_view<V, Pattern>::outer_iterator {
    private:
      // ...
    public:
      // ...
      using difference_type = iter_difference_t<iterator_t<Base>>range_difference_t<Base>;
      // ...
    };
  }
```

[...]

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

```
  namespace std::ranges {
    template<class V, class Pattern>
    template<bool Const>
    struct split_view<V, Pattern>::inner_iterator { // exposition only
    private:
      // ...
    public:
      // ...
      using value_type        = iter_value_t<iterator_t<Base>>range_value_t<Base>;
      using difference_type   = iter_difference_t<iterator_t<Base>>range_difference_t<Base>;
      // ...
    };
  }
```

[...]

23.7.13 Counted view [range.counted]

[...]

23.7.14 Common view [range.common]

[...]

23.7.15 Reverse view [range.reverse]

[...]

23.7.16 Istream view [range.istream]

23.7.16.1 Overview [range.istream.overview]

1 `basic_istream_view` models an `InputRange` and reads (using `operator>>`) successive elements from the input stream for which it was constructed.

2 If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to `default_sentinel`. The default constructor for `basic_istream_range` will always yield iterators equal to `default_sentinel`.

[*Example*:

```
auto ints = istringstream{"0 1  2   3     4"};
copy(istream_view(ints), ostream_iterator<int>{cout, "-"});
// prints 0-1-2-3-4-
```

— *end example*]

[*Note*: Although there are similarities in usage between `istream_iterator` and `basic_istream_view`, there are notable design differences and implementation differences between the two. Specifically, iterators to `basic_istream_view` do not model `EqualityComparable`, and so a default-constructed cannot be used to denote the past-the-end iterator. — *end note*]

23.7.16.2 Class template `basic_istream_view` [range.istream.view]

```
namespace std::ranges {
  template<class T, class U>
    concept stream-extractable = // exposition only
      requires(basic_istream<CharT, Traits>& is, T& t) {
         {is >> t} -> Same<basic_istream<CharT, Traits>>;
      };
```

1 Let `is` be an object of type `basic_istream<CharT, Traits>` and `t` be an object of type `T`.

(1.1) — `addressof(is) == addressof(is >> t)`.

```
  template<Movable Val, class CharT, class Traits>
    requires DefaultConstructible<Val> &&
      stream-extractable<CharT, Traits>
  class basic_istream_view : public view_interface<basic_istream_view<Val, CharT, Traits>> {
  public:
    basic_istream_view() = default;
    constexpr explicit basic_istream_view(basic_istream<CharT, Traits>& stream);

    constexpr auto begin();
    constexpr default_sentinel end() const noexcept;
  private:
    struct iterator;                          // exposition only
    basic_istream<CharT, Traits>* stream_; // exposition only
    Val object_{};                            // exposition only
  };
}
```

`constexpr explicit basic_istream_view(basic_istream<CharT, Traits>& stream);`

2 *Effects:* Initializes `stream_` to `addressof(stream)`.

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

3 *Effects:* Equivalent to:

```
*stream_ >> object_;
return iterator{*this};
```

```
constexpr default_sentinel end() const noexcept;
```

4 *Returns:* `default_sentinel`.

23.7.16.3 Class template `basic_istream_view::iterator` [range.istream.iterator]

```
namespace std::ranges {
  template<class Val, class CharT, class Traits>
  class basic_istream_view<Val, CharT, Traits>::iterator { // exposition only
  public:
    using iterator_category = input_iterator_tag;
    using difference_type = ptrdiff_t;
    using value_type = Val;

    iterator() = default;
    constexpr explicit iterator(basic_istream_view& parent) noexcept;

    iterator& operator++();
    void operator++(int);

    Val& operator*() const;

    friend bool operator==(iterator x, default_sentinel);
    friend bool operator==(default_sentinel y, iterator x);
    friend bool operator!=(iterator x, default_sentinel y);
    friend bool operator!=(default_sentinel y, iterator x);
  private:
    basic_istream_view<Val, CharT, Traits>* parent_ = nullptr; // exposition only
  };

  template<Movable T, class CharT, class Traits>
  requires DefaultConstructible<Val> && stream-extractable<Val, CharT, Traits>
  basic_istream_view<T, CharT, Traits> istream_view(basic_istream<CharT, Traits>& s);
}
```

```
constexpr explicit iterator(basic_istream_view& parent) noexcept;
```

1 *Effects:* Initializes `parent_` with `addressof(parent_)`.

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

2 *Effects:* Equivalent to:

```
*parent_->stream >> parent_->object_;
return *this;
```

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

3 *Effects:* Equivalent to: `++*this;`.

```
Val& operator*() const;
```

4 *Effects:* Equivalent to: `return parent_->value_;`

```
friend bool operator==(iterator x, default_sentinel);
```

5 *Effects:* Equivalent to: `return !*x.parent_->stream_;`

```
friend bool operator==(default_sentinel y, iterator x);
```

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

```
friend bool operator!=(iterator x, default_sentinel y);
```

```
friend bool operator!=(default_sentinel y, iterator x);
```

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

```
template<Movable T, class CharT, class Traits>
  requires DefaultConstructible<Val> && stream-extractable<Val, CharT, Traits>
basic_istream_view<T, CharT, Traits> istream_view(basic_istream<CharT, Traits>& s);
```

8 *Effects:* Equivalent to: `return basic_istream_view<T, CharT, Traits>;`

23.7.17 Elements view [range.elements]

23.7.17.1 Overview [range.elements.overview]

1 `elements_view` takes a `View` of *tuple-like* values and a `size_t`, and produces a `View` with a value-type of the Nth element of the adapted `View`'s value-type.

[*Example*:

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

auto names = elements<0>{historical_figures};
for (auto&& name : names) {
  format("{} ", name); // prints Babbage Hamilton Lovelace Turing
}

auto birth_years = elements<1>{historical_figures};
for (auto&& born : birth_years) {
  format("{} ", born); // prints 1791 1936 1815 1912
}
```

— *end example*]

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

[*Example*:

```
auto names = keys_view{historical_figures};
for (auto&& name : names) {
  format("{} ", name); // prints Babbage Hamilton Lovelace Turing
}
```

— *end example*]

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

[*Example*:

```
auto is_even = [](const auto x) { return x % 2 == 0; };
format("{} ", count_if(values_view{historical_figures}, is_even)); // prints 1936 1912
```

— *end example*]

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

```
namespace std::ranges {
  template<class T, size_t N>
    concept tuple-like = // exposition only
      requires { typename tuple_size<T>; } &&
      Same<tuple_size_v<T>, size_t>; &&
      0 < tuple_size_v<T> &&
      requires(T& t, const T& c) {
        typename tuple_element_t<T, tuple_size_v<T> - 1>;
        { get<N>(t) } -> Same<tuple_element_t<T, 0>&>;
        { get<N>(c) } -> Same<tuple_element_t<const T, 0>&>;
```

```
  };
```

1 Calls to `get` are looked up by argument-dependent lookup only, and ignore non-ADL lookup.

[Editor's note: *tuple-like* is in need of a lot of work.]

```
template<InputRange R, size_t N>
  requires View<R> && tuple-like<range_value_t<R>, N> &&
    tuple-like<remove_reference_t<range_reference_t<R>>, N>
class elements_view : public view_interface<elements_view<R, N>> {
public:
  elements_view() = default;
  constexpr explicit elements_view(R base);

  constexpr R base() const noexcept;

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

  constexpr auto end() requires (!simple-view<const R>);
  constexpr auto end() const requires simple-view<const R>;

  constexpr auto size() requires (SizedRange<R> && !simple-view<const R>);
  constexpr auto size() const
    requires (SizedRange<const R> && simple-view<const R>);
private:
  template<bool> struct iterator; // exposition only
  template<bool> struct sentinel; // exposition only
  R base_{};                      // exposition only
};
}
```

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

2 *Effects:* Initializes `base_` with `base`.

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

3 *Effects:* Equivalent to: `return base_;`

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

4 *Effects:* Equivalent to: `return iterator<is_const_v<decltype(*this)>>(*this, begin(base_-))`;

```
constexpr auto end() requires (!simple-view<const R>);
constexpr auto end() const requires simple-view<const R>;
```

5 *Effects:* Equivalent to: `return @sentinel@<is_const_v<decltype(*this)>>(*this, end(base_-))`;

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

6 *Effects:* Equivalent to: `return size(base);`

23.7.17.3 Class template `elements_view::iterator` [range.elements_view.iterator]

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

```
    parent_t* parent_t = nullptr;  // exposition only
    iterator_t<base_t> current_; // exposition only
  public:
    using iterator_category = iterator_category_t<iterator_t<base>>;
    using value_type = remove_cvref_t<tuple_element_t<N, range_value_t<base_t>>>;
    using difference_type = range_difference_t<base_t>;

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

    constexpr iterator_t<base_t> base() const noexcept;

    constexpr decltype(auto) operator*() const;

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

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

    constexpr iterator operator+=() requires RandomAccessRange<base_t>;
    constexpr iterator operator-=() requires RandomAccessRange<base_t>;

    constexpr decltype(auto) operator[](difference_type n) requires RandomAccessRange<base_t>;

    constexpr bool operator==(const iterator& x, const iterator& y)
      requires EqualityComparable<base_t>;
    constexpr bool operator!=(const iterator& x, const iterator& y)
      requires EqualityComparable<base_t>;
    constexpr bool operator<(const iterator& x, const iterator& y)
      requires RandomAccessRange<base_t>;
    constexpr bool operator>(const iterator& x, const iterator& y)
      requires RandomAccessRange<base_t>;
    constexpr bool operator<=(const iterator& y, const iterator& y)
      requires RandomAccessRange<base_t>;
    constexpr bool operator>=(const iterator& x, const iterator& y)
      requires RandomAccessRange<base_t>;
    constexpr iterator operator+(const iterator& x, difference_type y)
      requires RandomAccessRange<base_t>;
    constexpr iterator operator+(difference_type x, const iterator& y)
      requires RandomAccessRange<base_t>;
    constexpr iterator operator-(const iterator& x, difference_type y)
      requires RandomAccessRange<base_t>;
    constexpr difference_type operator-(const iterator& x, iterator y)
      requires RandomAccessRange<base_t>;
  };
}
```

```
constexpr explicit iterator(parent_t& parent, iterator_t<base_t> current);
```

1 *Effects:* Initializes `parent_` with `addressof(parent)` and `current_` with `current`.

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

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

```
constexpr iterator_t<base_t> base() const noexcept;
```

3 *Effects:* Equivalent to: `return current;`

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

4 *Effects:* Equivalent to: `return std::get<N>(*current_);`

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

Effects: Equivalent to:

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

```
constexpr void operator++(int) requires (!ForwardRange<base_t>;
```

Effects: Equivalent to: `++current_;`

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

Effects: Equivalent to:

```
auto temp = *this;
++current_;
return temp;
```

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

Effects: Equivalent to:

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

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

Effects: Equivalent to:

```
auto temp = *this;
--current_;
return temp;
```

```
constexpr iterator operator+=() requires RandomAccessRange<base_t>;
```

Effects: Equivalent to: `current_ += n; return *this;`

```
constexpr iterator operator-=() requires RandomAccessRange<base_t>;
```

Effects: Equivalent to: `current_ -= n; return *this;`

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

Effects: Equivalent to: `return *(current_ + n);`

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

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

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

Effects: Equivalent to: `return !(x == y);`

```
constexpr bool operator<(const iterator& x, const iterator& y)
  requires RandomAccessRange<base_t>;
```

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

```
constexpr bool operator>(const iterator& x, const iterator& y)
  requires RandomAccessRange<base_t>;
```

Effects: Equivalent to: `return y < x;`

```
constexpr bool operator<=(const iterator& x, const iterator& y)
  requires RandomAccessRange<base_t>;
```

Effects: Equivalent to: `return !(y < x);`

```
constexpr bool operator>=(const iterator& x, const iterator& y)
  requires RandomAccessRange<base_t>;
```

Effects: Equivalent to: `return !(x < y);`

```
constexpr iterator operator+(const iterator& x, difference_type y)
  requires RandomAccessRange<base_t>;
```

19 *Effects:* Equivalent to: `return iterator{x} += y;`

```
constexpr iterator operator+(difference_type x, const iterator& y)
  requires RandomAccessRange<base_t>;
```

20 *Effects:* Equivalent to: `return y + x;`

```
constexpr iterator operator-(const iterator& x, difference_type y)
  requires RandomAccessRange<base_t>;
```

21 *Effects:* Equivalent to: `return x + -y;`

```
constexpr difference_type operator-(const iterator& x, iterator y)
  requires RandomAccessRange<base_t>;
```

22 *Effects:* Equivalent to: `return x.current_ - y.current_;`

23.7.17.4 Class template `elements_view::sentinel` [range.elements_view.sentinel]

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

    sentinel_t<base_t> end_{}; // exposition only
    friend sentinel<!Const>;   // exposition only
  public:
    sentinel() = default;
    constexpr explicit sentinel(sentinel<base_t> end);
    constexpr explicit sentinel(sentinel<!Const> i)
      requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<const R>>;

    constexpr sentinel_t<base_t> base() const;

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

    constexpr friend range_difference_t<base_t>
      operator-(const iterator<Const>& x, const sentinel& y)
        requires SizedSentinel<sentinel_t<base_t>, base_t>>;
    constexpr friend range_difference_t<base_t>
      operator-(const sentinel& x, const iterator<Const>& y)
        requires SizedSentinel<sentinel_t<base_t>, base_t>>;
  };
}
```

```
constexpr explicit sentinel(sentinel<base_t> end);
```

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

```
constexpr explicit sentinel(sentinel<!Const> i)
  requires Const && ConvertibleTo<sentinel_t<R>, sentinel_t<const R>>;
```

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

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

3 *Effects:* Equivalent to: `return base_;`

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

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

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

5 *Effects:* Equivalent to: `return y == x;`

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

6 *Effects:* Equivalent to: `return !(x == y);`

```
constexpr friend range_difference_t<base_t>
  operator-(const iterator<Const>& x, const sentinel& y)
    requires SizedSentinel<sentinel_t<base_t, base_t>>;
```

7 *Effects:* Equivalent to: `return x.current_ - y.end_;`

```
constexpr friend range_difference_t<base_t>
  operator-(const sentinel& x, const iterator<Const>& y)
    requires SizedSentinel<sentinel_t<base_t, base_t>>;
```

8 *Effects:* Equivalent to: `return -(y - x);`

23.7.17.5 `view::elements` [range.elements.adaptor]

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

23.7.17.6 `view::keys` [range.keys.adaptor]

The name `view::keys` denotes a range adaptor object (??). For some subexpression `E`, the expression `view::keys(E)` is expression-equivalent to `elements_view<decltype((E)), 0>E`.

23.7.17.7 `view::values` [range.values.adaptor]

The name `view::values` denotes a range adaptor object (??). For some subexpression `E`, the expression `view::keys(E)` is expression-equivalent to `elements_view<decltype((E)), 1>E`.

24 Algorithms library [algorithms]

24.1 General [algorithms.general]

[...]

24.2 Header <algorithm> synopsis [algorthim.syn]

[Editor's note: All changes in this chapter are to accommodate the new associated range types introduced in this document.]

```
namespace std {
  // ...
  namespace ranges {
    // ...
    template<InputRange R, class T, class Proj = identity>
      requires IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
      constexpr ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R>
        count(R&& r, const T& value, Proj proj = {});
    // ...
    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R>
        count_if(R&& r, Pred pred, Proj proj = {});
  }
  // ...
  namespace ranges {
    // ...
    template<ForwardRange R, class T, class Pred = ranges::equal_to,
             class Proj = identity>
      requires IndirectlyComparable<iterator_t<R>, const T*, Pred, Proj>
      constexpr safe_subrange_t<R>
        search_n(R&& r, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> count,
                 const T& value, Pred pred = {}, Proj proj = {});
  }
  // ...
  namespace ranges {
    // ...
    template<InputRange R, WeaklyIncrementable O, class Proj = identity,
             IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>
      requires IndirectlyCopyable<iterator_t<R>, O> &&
               (ForwardIterator<iterator_t<R>> ||
                (InputIterator<O> && Same<~~iter_value_t<iterator_t<R>>~~range_value_t<R>, iter_value_t<O>>) ||
                IndirectlyCopyableStorable<iterator_t<R>, O>)
      constexpr unique_copy_result<safe_iterator_t<R>, O>
        unique_copy(R&& r, O result, C comp = {}, Proj proj = {});
  }
  // ...
  namespace ranges {
    // ...
    template<InputRange R, WeaklyIncrementable O, class Gen>
      requires (ForwardRange<R> || RandomAccessIterator<O>) &&
               IndirectlyCopyable<iterator_t<R>, O> &&
               UniformRandomBitGenerator<remove_reference_t<Gen>>
    sample_result<I, O>
    sample(R&& r, O out, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> n, Gen&& g);
  }
  // ...
  namespace ranges {
    // ...
    template<ForwardRange R>
```

```
      requires Permutable<iterator_t<R>>
    constexpr safe_subrange_t<R> shift_left(R&& r, iter_difference_t<iterator_t<R>>range_difference_t<R> n);
  }
  // ...
  namespace ranges {
    // ...
    template<ForwardRange R>
      requires Permutable<iterator_t<R>>
    constexpr safe_subrange_t<Rng> shift_right(R&& r, iter_difference_t<iterator_t<R>>range_difference_t<R> n
  }
  // ...
  namespace ranges {
    // ...
    template<InputRange R, class Proj = identity,
             IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
    constexpr iter_value_t<iterator_t<R>>range_value_t<R>
      min(R&& r, Comp comp = {}, Proj proj = {});
  }
  // ...
  namespace ranges {
    // ...
    template<InputRange R, class Proj = identity,
             IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>range_value_t<R>*>
    constexpr iter_value_t<iterator_t<R>>range_value_t<R>
      max(R&& r, Comp comp = {}, Proj proj = {});
  }
  // ...
  namespace ranges {
    // ...
    template<InputRange R, class Proj = identity,
             IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>range_value_t<R>*>
    constexpr minmax_result<iter_value_t<iterator_t<R>>range_value_t<R>>
      minmax(R&& r, Comp comp = {}, Proj proj = {});
  }
  // ...
}
```

24.3 Count [alg.count]

```
namespace ranges {
  // ...
  template<InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
    constexpr iter_difference_t<iterator_t<R>>range_difference_t<R>
      count(R&& r, const T& value, Proj proj = {});
  // ...
  template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  constexpr iter_difference_t<iterator_t<R>>range_differnece_t<R>
    count_if(R&& r, Pred pred, Proj proj = {});
}
```

24.4 Search [alg.search]

```
// ...
namespace ranges {
  template<ForwardRange R, class T, class Pred = ranges::equal_to,
           class Proj = identity>
     requires IndirectlyComparable<iterator_t<R>, const T*, Pred, Proj>
     constexpr safe_subrange_t<R>
       search_n(R&& r, iter_difference_t<iterator_t<R>>range_difference_t<R> count,
                const T& value, Pred pred = {}, Proj proj = {});
```

```
  }
```

24.5 Unique copy [alg.unique_copy]

```
  namespace ranges {
    // ...
    template<InputRange R, WeaklyIncrementable O, class Proj = identity,
             IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>
      requires IndirectlyCopyable<iterator_t<R>, O> &&
               (ForwardIterator<iterator_t<R>> ||
                (InputIterator<O> && Same<~~iter_value_t<iterator_t<R>>~~range_value_t<R>, iter_value_t<O>>) ||
                IndirectlyCopyableStorable<iterator_t<R>, O>)
      constexpr unique_copy_result<safe_iterator_t<R>, O>
        unique_copy(R&& r, O result, C comp = {}, Proj proj = {});
  }
```

[...]

24.6 Sample [alg.random.sample]

```
// ...
namespace ranges {
  // ...
  template<InputRange R, WeaklyIncrementable O, class Gen>
    requires (ForwardRange<R> || RandomAccessIterator<O>) &&
             IndirectlyCopyable<iterator_t<R>, O> &&
             UniformRandomBitGenerator<remove_reference_t<Gen>>
    sample_result<I, O>
  sample(R&& r, O out, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> n, Gen&& g);
}
```

[...]

24.7 Shift [alg.shift]

```
// ...
namespace ranges {
  // ...
  template<ForwardRange R>
    requires Permutable<iterator_t<R>>
  constexpr safe_subrange_t<R> shift_left(R&& r, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> n);
}
```

[...]

```
// ...
namespace ranges {
  // ...
  template<ForwardRange R>
    requires Permutable<iterator_t<R>>
  constexpr safe_subrange_t<Rng> shift_right(R&& r, ~~iter_difference_t<iterator_t<R>>~~range_difference_t<R> n);
}
```

[...]

24.8 Minimum and maximum [alg.min.max]

```
  namespace ranges {
    // ...
    template<InputRange R, class Proj = identity,
             IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>*>
    constexpr ~~iter_value_t<iterator_t<R>>~~range_value_t<R>
      min(R&& r, Comp comp = {}, Proj proj = {});
  }
```

[...]

```
  // ...
  namespace ranges {
    // ...
    template<InputRange R, class Proj = identity,
             IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, ~~iter_value_t<iterator_t<R>>~~range_value_t<R>*>
    constexpr ~~iter_value_t<iterator_t<R>>~~range_value_t<R>
      max(R&& r, Comp comp = {}, Proj proj = {});
  }
```

[...]

```
  // ...
  namespace ranges {
    // ...
    template<InputRange R, class Proj = identity,
             IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requires IndirectlyCopyableStorable<iterator_t<R>, ~~iter_value_t<iterator_t<R>>~~range_value_t<R>*>
    constexpr minmax_result<~~iter_value_t<iterator_t<R>>~~range_value_t<R>>
      minmax(R&& r, Comp comp = {}, Proj proj = {});
  }
```

[...]