Document Number: P0896
Date: 2018-02-12
Reply to: Eric Niebler
eric.niebler@gmail.com

Merging the Ranges TS

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

Contents

Contents ii

1 Scope 1

2 Normative References 2

3 Terms and Definitions 3

4 General Principles 4
- 4.1 Goals 4
- 4.2 Rationale 4
- 4.3 Risks 4
- 4.4 Methodology 5
- 4.5 Implementation compliance 5
- 4.6 Namespaces, headers, and modifications to standard classes 5

21 Standard Library, Version 2 6
- 21.1 Library introduction 6
- 21.2 General utilities library 10
- 21.3 Iterators library 19
- 21.4 Ranges library 86
- 21.5 Algorithms library 95
- 21.6 Numerics library 146

A Acknowledgements 147

Bibliography 148

Index 149

Index of library names 150

1 Scope [intro.scope]

"Eventually, all things merge into one, and a river runs through it."

—*Norman Maclean*

1 This document proposes to merge the ISO/IEC TS 21425:2017, aka the Ranges TS, into the working draft. This document is intended to be taken in conjunction with P0898, a paper which proposes importing the definitions of the Ranges TS's Concepts library (Clause 7) into namespace `std`.

2 Normative 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, *Programming Languages - C++*

(1.2) — ISO/IEC TS 21425:2017, *Technical Specification - C++ Extensions for Ranges*

ISO/IEC 14882 is herein called the *C++ Standard* and ISO/IEC TS 21425:2017 is called the *Ranges TS*.

3 Terms and Definitions [intro.defs]

[Editor's note: The following definitions are hereby proposed for subclause [definitions] of ISO/IEC 14882.]

For the purposes of this document, the terms and definitions given in ISO/IEC 14882 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at http://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1 [std2.defns.projection]

projection

⟨function object argument⟩ transformation which an algorithm applies before inspecting the values of elements

[*Example:*

```
std::pair<int, const char*> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std2::sort(pairs, std::less<>{}, [](auto const& p) { return p.first; });
```

sorts the pairs in increasing order of their `first` members:

```
{{0, "baz"}, {1, "bar"}, {2, "foo"}}
```

— *end example*]

4 General Principles [intro]

4.1 Goals [intro.goals]

1 The primary goal of this proposal is to deliver high-quality, constrained generic Standard Library components at the same time that the language gets support for such components.

4.2 Rationale [intro.rationale]

1 The best, and arguably only practical way to achieve the goal stated above is by incorporating the Ranges TS into the working paper. The sooner we can agree on what we want "`Iterator`" and "`Range`" to mean going forward (for instance), and the sooner users are able to rely on them, the sooner we can start building and delivering functionality on top of those fundamental abstractions. (For example, see "P0789: Range Adaptors and Utilities" ([4]).)

2 The cost of not delivering such a set of Standard Library concepts and algorithms is that users will either do without or create a babel of mutually incompatible concepts and algorithms, often without the rigour followed by the Ranges TS. The experience of the authors and implementors of the Ranges TS is that getting concept definitions and algorithm constraints right is *hard*. The Standard Library should save its users from needless heartache.

4.3 Risks [intro.risks]

1 Shipping constrained components from the Ranges TS in the C++20 timeframe is not without risk. As of the time of writing (February 1, 2018), no major Standard Library vendor has shipped an implementation of the Ranges TS. Two of the three major compiler vendors have not even shipped an implementation of the concepts language feature. Arguably, we have not yet gotten the usage experience for which all Technical Specifications are intended.

2 On the other hand, the components of Ranges TS have been vetted very thoroughly by the range-v3 ([3]) project, on which the Ranges TS is based. There is no part of the Ranges TS – concepts included – that has not seen extensive use via range-v3. (The concepts in range-v3 are emulated with high fidelity through the use of generalized SFINAE for expressions.) As an Open Source project, usage statistics are hard to come by, but the following may be indicitive:

(2.1) — The range-v3 GitHub project has over 1,400 stars, over 120 watchers, and 145 forks.

(2.2) — It is cloned on average about 6,000 times a month.

(2.3) — A GitHub search, restricted to C++ files, for the string "`range/v3`" (a path component of all of range-v3's header files), turns up over 7,000 hits.

3 Lacking true concepts, range-v3 cannot emulate concept-based function overloading, or the sorts of constraints-checking short-circuit evaluation required by true concepts. For that reason, the authors of the Ranges TS have created a reference implementation: CMCSTL2 ([1]) using true concepts. To this reference implementation, the authors ported all of range-v3's tests. These exposed only a handful of concepts-specific bugs in the components of the Ranges TS (and a great many more bugs in compilers). Those improvements were back-ported to range-v3 where they have been thoroughly vetted over the past 2 years.

4 In short, concern about lack of implementation experience should not be a reason to withhold this important Standard Library advance from users.

4.4 Methodology [intro.methedology]

1 The contents of the Ranges TS, Clause 7 (“Concepts library”) are proposed for namespace `std` by P0898, “Standard Library Concepts” ([2]). Additionally, P0898 proposes the `identity` function object (ISO/IEC TS 21425:2017 §8.3.3) and the `common_reference` type trait (ISO/IEC TS 21425:2017 §8.4.3) for namespace `std`, and the `swap` customization point object for namespace `std2`. The changes proposed by the Ranges TS to `common_type` are merged into the working paper (also by P0898). The “`invoke`” function and the “`swappable`” type traits (e.g., `is_swappable_with`) already exist in the text of the working paper, so they are omitted here.

2 The salient, high-level features of this proposal are as follows:

(2.1) — The remaining library components in the Ranges TS are proposed for namespace `::std2::`.

(2.2) — The text of the Ranges TS is rebased on the latest working draft.

(2.3) — Structurally, this paper proposes to relocate the existing library clauses of the working draft (20-33) down one level under a new clause 20, provisionally titled “Standard Library, Version 1”. No stable names are changed.

(2.4) — The Concepts Library clause, proposed by P0898, is located in that paper between the “Language Support Library” and the “Diagnostics library”. In the organization proposed by this paper, that places it as subclause 20.3. This paper refers to it as such.

(2.5) — We additionally propose that a new clause 21 be created, provisionally titled “Standard Library, Version 2”, and the following clauses of the Ranges TS should be made subclauses of this clause: 6, 8-12.

(2.6) — Where the text of the Ranges TS needs to be updated, the text is presented with change markings: ~~red strikethrough~~ for removed text and blue underline for added text.

(2.7) — The stable names of everything in the Ranges TS, clauses 6, 8-12 are changed by preprending “`std2.`”. References are updated accordingly.

(2.8) — The headers of the Ranges TS are renamed from `<experimental/ranges/`*`foo`*`>` to `<std2/`*`foo`*`>`.

4.5 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in 4.1 in the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

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

1 Since the library components described in this document are constrained versions of facilities already found within namespace `std`, we propose to define everything within namespace `::std2::v1`, where `v1` is an inline namespace.

2 Unless otherwise specified, references to entities described in this document are assumed to be qualified with `::std2::`, and references to entities described in the current working draft of the International Standard, including the entities of the “Concepts library” proposed by P0898, are assumed to be qualified with `::std::`.

21 Standard Library, Version 2 [std2]

21.1 Library introduction [std2.library]

21.1.1 General [std2.library.general]

1 This Clause describes the contents of the ~~*Ranges library*~~ *Standard Library, Version 2* , how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

2 ~~Clause~~Section 21.1.3~~,~~ and ~~C~~Subclauses 21.2 through 21.6~~, and Annex ??~~ specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.

3 Detailed specifications for each of the components in the library are in ~~C~~Subclauses 21.2–21.6, as shown in Table 1.

Table 1 — Library categories

Clause	Category
~~??~~	~~Concepts library~~
21.2	General utilities library
21.3	Iterators library
21.4	Ranges library
21.5	Algorithms library
21.6	Numerics library

4 ~~The concepts library (Clause ??) describes library components that C++ programs may use to perform compile-time validation of template parameters and perform function dispatch based on properties of types.~~

5 The general utilities library (~~C~~Subclause 21.2) includes components used by other library elements and components used as infrastructure in C++ programs, such as function objects.

6 The iterators library (~~C~~Subclause 21.3) describes components that C++ programs may use to perform iterations over containers (Clause 20.8), streams (20.12.7), stream buffers (20.12.6), and ranges (21.4).

7 The ranges library (~~C~~Subclause 21.4) describes components for dealing with ranges of elements.

8 The algorithms library (~~C~~Subclause 21.5) describes components that C++ programs may use to perform algorithmic operations on containers (Clause 20.8) and other sequences.

9 The numerics library (~~C~~Subclause 21.6) provides concepts that are useful to constrain numeric algorithms.

21.1.2 Method of description (Informative) [std2.description]

1 This subclause describes the conventions used to specify the ~~Ranges library~~Standard Library, Version 2. 21.1.2.1 describes the structure of the normative ~~C~~Subclauses 21.2 through 21.6~~ and Annex ??~~. 21.1.2.2 describes other editorial conventions.

21.1.2.1 Structure of each clause [std2.structure]

21.1.2.1.1 Elements [std2.structure.elements]

1 Each library ~~c~~subclause contains the following elements, as applicable:[1]

1) To save space, items that do not apply to a ~~C~~Subclause are omitted. For example, if a ~~C~~Subclause does not specify any requirements, there will be no "Requirements" ~~subclause~~section.

(1.1) — Summary

(1.2) — Requirements

(1.3) — Detailed specifications

21.1.2.1.2 Summary [std2.structure.summary]

1 The Summary provides a synopsis of the category, and introduces the first-level ~~subclauses~~sections. Each ~~subclause~~section also provides a summary, listing the headers specified in the ~~subclause~~section and the library entities provided in each header.

2 Paragraphs labeled "Note(s):" or "Example(s):" are informative, other paragraphs are normative.

3 The contents of the summary and the detailed specifications include:

(3.1) — macros

(3.2) — values

(3.3) — types

(3.4) — classes and class templates

(3.5) — functions and function templates

(3.6) — objects

(3.7) — concepts

21.1.2.1.3 Requirements [std2.structure.requirements]

1 Requirements describe constraints that shall be met by a C++ program that extends the ~~Ranges library~~Standard Library, Version 2. Such extensions are generally one of the following:

(1.1) — Template arguments

(1.2) — Derived classes

(1.3) — Containers, iterators, and algorithms that meet an interface convention or satisfy a concept

2 Interface convention requirements are stated as generally as possible. Instead of stating "class X has to define a member function `operator++()`," the interface requires "for any object `x` of class `X`, `++x` is defined." That is, whether the operator is a member is unspecified.

3 Requirements are stated in terms of concepts (~~Concepts TS [dcl.spec.concept]~~17). Concepts are stated in terms of well-defined expressions that define valid terms of the types that satisfy the concept. For every set of well-defined expression requirements there is a named concept that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (~~C~~Subclause 21.5) that uses the well-defined expression requirements is described in terms of the valid expressions for its formal type parameters.

4 Template argument requirements are sometimes referenced by name. See 20.1.4.2.1.

5 In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.[2]

6 Required operations of any concept defined in this document need not be total functions; that is, some arguments to a required operation may result in the required semantics failing to be satisfied. [*Example:*

2) Although in some cases the code given is unambiguously the optimum implementation.

The required `<` operator of the `StrictTotallyOrdered` concept (20.3.4.4) does not meet the semantic requirements of that concept when operating on NaNs. — *end example*] This does not affect whether a type satisfies the concept.

7 A declaration may explicitly impose requirements through its associated constraints (~~Concepts TS~~ [temp.constr.decl]). When the associated constraints refer to a concept (~~Concepts TS~~ [~~dcl.spec~~temp.concept]), additional semantic requirements are imposed on the use of the declaration.

21.1.2.1.4 Detailed specifications [std2.structure.specifications]

1 The detailed specifications of each entity defined in Clauses 21.2–21.6 follow the conventions established by .

21.1.2.2 Other conventions [std2.conventions]

1 This subclause describes several editorial conventions used to describe the contents of the ~~Ranges library~~Standard Library, Version 2. These conventions are for describing member functions (21.1.2.2.1), and private members (21.1.2.2.2).

21.1.2.2.1 Functions within classes [std2.functions.within.classes]

1 This ~~document~~Clause follows the same conventions as specified in .

21.1.2.2.2 Private members [std2.objects.within.classes]

1 This ~~document~~Clause follows the same conventions as specified in .

21.1.3 Library-wide requirements [std2.requirements]

1 This subclause specifies requirements that apply to the entire ~~Ranges library~~Standard Library, Version 2. ~~C~~Subclauses 21.2 through 21.6~~ and Annex ??~~ specify the requirements of individual entities within the library.

2 Requirements specified in terms of interactions between threads do not apply to programs having only a single thread of execution.

3 Within this subclause, 21.1.3.1 describes the library's contents and organization, 21.1.3.3 describes how well-formed C++ programs gain access to library entities, 21.1.3.4 describes constraints on well-formed C++ programs, and 21.1.3.5 describes constraints on conforming implementations.

21.1.3.1 Library contents and organization [std2.organization]

1 21.1.3.1.1 describes the entities and macros defined in the ~~Ranges library~~Standard Library, Version 2.

21.1.3.1.1 Library contents [std2.contents]

1 The ~~Ranges library~~Standard Library, Version 2 provides definitions for the entities and macros specified in the ~~Ranges library~~Standard Library, Version 2 headers (21.1.3.2).

2 All library entities are defined within an inline namespace `v1` within the namespace ~~`std::experimental::ranges`~~`::std2` or namespaces nested within namespace ~~`std::experimental::ranges::v1`~~`::std2::v1`. It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.

21.1.3.2 Headers [std2.headers]

1 Each element of the ~~Ranges library~~Standard Library, Version 2 is declared or defined (as appropriate) in a header.

2 The ~~Ranges library~~Standard Library, Version 2 provides the ~~Ranges library~~Standard Library, Version 2 headers, shown in Table 2.

Table 2 — ~~Ranges TS library~~Standard Library, Version 2 headers

<~~experimental/ranges~~std2/algorithm>	<~~experimental/ranges~~std2/range>
~~<experimental/ranges/concepts>~~	<~~experimental/ranges~~std2/tuple>
<~~experimental/ranges~~std2/functional>	~~<experimental/ranges/type_traits>~~
<~~experimental/ranges~~std2/iterator>	<~~experimental/ranges~~std2/utility>
<~~experimental/ranges~~std2/random>	

21.1.3.3 Using the library [std2.using]

21.1.3.3.1 Overview [std2.using.overview]

1 This section describes how a C++ program gains access to the facilities of the ~~Ranges library~~Standard Library, Version 2. 21.1.3.3.2 describes effects during translation phase 4, while 21.1.3.3.3 describes effects during phase 8 ().

21.1.3.3.2 Headers [std2.using.headers]

The entities in the ~~Ranges library~~Standard Library, Version 2 are defined in headers, the use of which is governed by the same requirements as specified in .

21.1.3.3.3 Linkage [std2.using.linkage]

1 Entities in the C++ standard library have external linkage (). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage ().

21.1.3.4 Constraints on programs [std2.constraints]

21.1.3.4.1 Overview [std2.constraints.overview]

1 This section describes restrictions on C++ programs that use the facilities of the ~~Ranges library~~Standard Library, Version 2. The following subclauses specify constraints on the program's use of ~~Ranges library~~Standard Library, Version 2 classes as base classes (21.1.3.4.2) and other constraints.

21.1.3.4.2 Derived classes [std2.derived.classes]

1 Virtual member function signatures defined for a base class in the ~~Ranges library~~Standard Library, Version 2 may be overridden in a derived class defined in the program (13.3).

21.1.3.4.3 Other functions [std2.res.on.functions]

1 In certain cases (operations on types used to instantiate ~~Ranges library~~Standard Library, Version 2 template components), the ~~Ranges library~~Standard Library, Version 2 depends on components supplied by a C++ program. If these components do not meet their requirements, this document places no requirements on the implementation.

2 In particular, the effects are undefined if an incomplete type (6.7) is used as a template argument when instantiating a template component or evaluating a concept, unless specifically allowed for that component.

21.1.3.4.4 Function arguments [std2.res.on.arguments]

1 The constraints on arguments passed to C++ standard library function as specified in also apply to arguments passed to functions in the ~~Ranges library~~Standard Library, Version 2.

21.1.3.4.5 Library object access [std2.res.on.objects]

1 The constraints on object access by C++ standard library functions as specified in also apply to object access by functions in the ~~Ranges library~~Standard Library, Version 2.

21.1.3.4.6 Requires paragraph [std2.res.on.required]

1 Violation of the preconditions specified in a function's *Requires:* paragraph results in undefined behavior unless the function's *Throws:* paragraph specifies throwing an exception when the precondition is violated.

21.1.3.4.7 Semantic requirements [std2.res.on.requirements]

1 If the semantic requirements of a declaration's constraints (21.1.2.1.3) are not satisfied at the point of use, the program is ill-formed, no diagnostic required.

21.1.3.5 Conforming implementations [std2.conforming]

1 The constraints upon, and latitude of, implementations of the ~~Ranges library~~Standard Library, Version 2 follow the same constraints and latitudes for implementations of the C++ standard library as specified in 20.1.5.5.

[Editor's note: Remove section [customization.point.object] "Customization Point Objects". They are introduced by P0898.]

21.2 General utilities library [std2.utilities]

21.2.1 General [std2.utilities.general]

1 This ~~C~~Subclause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the ~~Ranges library~~Standard Library, Version 2. These utilities are summarized in Table 3.

Table 3 — General utilities library summary

	Subclause	Header(s)
21.2.2	Utility components	`<`~~`experimental/ranges`~~`std2/utility>`
21.2.3	Function objects	`<`~~`experimental/ranges`~~`std2/functional>`
~~??~~	~~Type traits~~	~~`<type_traits>`~~
21.2.4	Tagged tuple-like types	`<`~~`experimental/ranges`~~`std2/utility>` & `<`~~`experimental/ranges`~~`std2/tuple>`

21.2.2 Utility components [std2.utility]

1 This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header <~~experimental/ranges~~std2/utility> synopsis

2 The header <~~experimental/ranges~~std2/utility> defines several types, function templates, and concepts that are described in this ~~C~~Subclause. It also defines the templates `tagged` and `tagged_pair` and various function templates that operate on `tagged_pair` objects.

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  // ??, swap:
  namespace {
    constexpr unspecified swap = unspecified;
  }

  // 21.2.2.1, exchange:
  template <MoveConstructible T, class U=T>
    requires Assignable<T&, U>
  constexpr T exchange(T& obj, U&& new_val) noexcept(see below);
```

```
// 21.2.4.2, struct with named accessors
template <class T>
concept bool TagSpecifier = see below;

template <class F>
concept bool TaggedType = see below;

template <class Base, TagSpecifier... Tags>
  requires sizeof...(Tags) <= tuple_size_v<Base>::value
struct tagged;

// 21.2.4.4, tagged pairs
template <TaggedType T1, TaggedType T2> using tagged_pair = see below;

template <TagSpecifier Tag1, TagSpecifier Tag2, class T1, class T2>
constexpr see below make_tagged_pair(T1&& x, T2&& y);
}}}}

namespace std {
  // 21.2.4.3, tuple-like access to tagged
  template <class Base, class... Tags>
  struct tuple_size<experimental::ranges::std2::tagged<Base, Tags...>>;

  template <size_t N, class Base, class... Tags>
  struct tuple_element<N, experimental::ranges::std2::tagged<Base, Tags...>>;
}
```

[Editor's note: Section [utility.swap] "swap" is moved to P0898.]

21.2.2.1 `exchange` [std2.utility.exchange]

```
template <MoveConstructible T, class U=T>
  requires Assignable<T&, U>
constexpr T exchange(T& obj, U&& new_val) noexcept(see below);
```

1 *Effects:* Equivalent to:

```
T old_val = std::move(obj);
obj = std::forward<U>(new_val);
return old_val;
```

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

```
is_nothrow_move_constructible_v<T>::value &&
is_nothrow_assignable_v<T&, U>::value
```

21.2.3 Function objects [std2.function.objects]

1 **Header `<~~experimental/ranges~~std2/functional>` synopsis**

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  // ??, invoke:
  template <class F, class... Args>
  result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

  // 21.2.3.1, comparisons:
  template <class T = void>
    requires see below
```

```
    struct equal_to;

    template <class T = void>
      requires see below
    struct not_equal_to;

    template <class T = void>
      requires see below
    struct greater;

    template <class T = void>
      requires see below
    struct less;

    template <class T = void>
      requires see below
    struct greater_equal;

    template <class T = void>
      requires see below
    struct less_equal;

    template <> struct equal_to<void>;
    template <> struct not_equal_to<void>;
    template <> struct greater<void>;
    template <> struct less<void>;
    template <> struct greater_equal<void>;
    template <> struct less_equal<void>;

    // ??, identity:
    struct identity;
  }}}}
```

[Editor's note: Section [func.invoke] "Function template `invoke`" is intentionally omitted.]

21.2.3.1 Comparisons [std2.comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language (8.5.9, 8.5.10).

2 In this section, *`BUILTIN_PTR_CMP`*`(T,` *op*`, U)` for types `T` and `U` and where *op* is an equality (8.5.10) or relational operator (8.5.9) is a boolean constant expression. *`BUILTIN_PTR_CMP`*`(T,` *op*`, U)` is `true` if and only if *op* in the expression `declval<T>()` *op* `declval<U>()` resolves to a built-in operator comparing pointers.

3 There is an implementation-defined strict total ordering over all pointer values of a given type. This total ordering is consistent with the partial order imposed by the builtin operators `<`, `>`, `<=`, and `>=`.

```
template <class T = void>
  requires EqualityComparable<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, ==, const T&)
struct equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

4 `operator()` has effects equivalent to: `return equal_to<>(x, y);`

```
template <class T = void>
```

```
  requires EqualityComparable<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, ==, const T&)
struct not_equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

5 `operator()` has effects equivalent to: `return !equal_to<>(x, y);`

```
template <class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct greater {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

6 `operator()` has effects equivalent to: `return less<>(y, x);`

```
template <class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct less {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

7 `operator()` has effects equivalent to: `return less<>(x, y);`

```
template <class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct greater_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

8 `operator()` has effects equivalent to: `return !less<>(x, y);`

```
template <class T = void>
  requires StrictTotallyOrdered<T> || Same<T, void> || BUILTIN_PTR_CMP(const T&, <, const T&)
struct less_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

9 `operator()` has effects equivalent to: `return !less<>(y, x);`

```
template <> struct equal_to<void> {
  template <class T, class U>
    requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
  constexpr bool operator()(T&& t, U&& u) const;

  typedef unspecified  is_transparent;
};
```

10 *Requires:* If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator `==` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving (20.3.1.1).

11 *Effects:*

(11.1) — If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator `==` comparing pointers of type `P`: returns `false` if either (the converted value of) `t` precedes `u` or `u` precedes `t` in the implementation-defined strict total order over pointers of type `P` and otherwise `true`.

(11.2) — Otherwise, equivalent to: `return std::forward<T>(t) == std::forward<U>(u);`

```
template <> struct not_equal_to<void> {
  template <class T, class U>
    requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
  constexpr bool operator()(T&& t, U&& u) const;

  typedef unspecified  is_transparent;
};
```

12 `operator()` has effects equivalent to:

```
return !equal_to<>{}(std::forward<T>(t), std::forward<U>(u));
```

```
template <> struct greater<void> {
  template <class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;

  typedef unspecified  is_transparent;
};
```

13 `operator()` has effects equivalent to:

```
return less<>{}(std::forward<U>(u), std::forward<T>(t));
```

```
template <> struct less<void> {
  template <class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;

  typedef unspecified  is_transparent;
};
```

14 *Requires:* If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator `<` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving (20.3.1.1). For any expressions `ET` and `EU` such that `decltype((ET))` is `T` and `decltype((EU))` is `U`, exactly one of `less<>{}(ET, EU)`, `less<>{}(EU, ET)` or `equal_to<>{}(ET, EU)` shall be `true`.

15 *Effects:*

(15.1) — If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator `<` comparing pointers of type `P`: returns `true` if (the converted value of) `t` precedes `u` in the implementation-defined strict total order over pointers of type `P` and otherwise `false`.

(15.2) — Otherwise, equivalent to: `return std::forward<T>(t) < std::forward<U>(u);`

```
template <> struct greater_equal<void> {
  template <class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;

  typedef unspecified  is_transparent;
};
```

16 `operator()` has effects equivalent to:

```
return !less<>{}(std::forward<T>(t), std::forward<U>(u));
```

```
template <> struct less_equal<void> {
  template <class T, class U>
    requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;

  typedef unspecified  is_transparent;
};
```

17 `operator()` has effects equivalent to:

```
return !less<>{}(std::forward<U>(u), std::forward<T>(t));
```

[Editor's note: The section [func.identity] of the Ranges TS is intentionally omitted.]

[Editor's note: Subsection [meta] "Metaprogramming and type traits" is intentionally omitted.]

21.2.4 Tagged tuple-like types [std2.taggedtup]

21.2.4.1 General [std2.taggedtup.general]

1 The library provides a template for augmenting a tuple-like type with named element accessor member functions. The library also provides several templates that provide access to `tagged` objects as if they were `tuple` objects (see 20.5.5.3.7).

21.2.4.2 Class template `tagged` [std2.taggedtup.tagged]

1 Class template `tagged` augments a tuple-like class type (e.g., `pair` (20.5.4), `tuple` (20.5.5)) by giving it named accessors. It is used to define the alias templates `tagged_pair` (21.2.4.4) and `tagged_tuple` (21.2.4.5).

2 In the class synopsis below, let i be in the range `[0,sizeof...(Tags))` and T_i be the i^{th} type in `Tags`, where indexing is zero-based.

```
// defined in header <~~experimental/ranges~~std2/utility>

namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <class T>
  concept ~~bool~~ TagSpecifier = implementation-defined;

  template <class F>
  concept ~~bool~~ TaggedType = implementation-defined;

  template <class Base, TagSpecifier... Tags>
    requires sizeof...(Tags) <= tuple_size_v<Base>~~::value~~
  struct tagged :
    Base, TAGGET(tagged<Base, Tags...>, Ti, i)... { // see below
    using Base::Base;
    tagged() = default;
    tagged(tagged&&) = default;
    tagged(const tagged&) = default;
    tagged &operator=(tagged&&) = default;
    tagged &operator=(const tagged&) = default;
    tagged(Base&&) noexcept(see below)
      requires MoveConstructible<Base>;
    tagged(const Base&) noexcept(see below)
      requires CopyConstructible<Base>;
    template <class Other>
      requires Constructible<Base, Other>
    constexpr tagged(tagged<Other, Tags...> &&that) noexcept(see below);
    template <class Other>
```

```
      requires Constructible<Base, const Other&>
    constexpr tagged(const tagged<Other, Tags...> &that);
    template <class Other>
      requires Assignable<Base&, Other>
    constexpr tagged& operator=(tagged<Other, Tags...>&& that) noexcept(see below);
    template <class Other>
      requires Assignable<Base&, const Other&>
    constexpr tagged& operator=(const tagged<Other, Tags...>& that);
    template <class U>
      requires Assignable<Base&, U> && !Same<decay_t<U>, tagged>
    constexpr tagged& operator=(U&& u) noexcept(see below);
    constexpr void swap(tagged& that) noexcept(see below)
      requires Swappable<Base>;
    friend constexpr void swap(tagged&, tagged&) noexcept(see below)
      requires Swappable<Base>;
  };
}}~~}}~~
```

3 A *tagged getter* is an empty trivial class type that has a named member function that returns a reference to a member of a tuple-like object that is assumed to be derived from the getter class. The tuple-like type of a tagged getter is called its *DerivedCharacteristic*. The index of the tuple element returned from the getter's member functions is called its *ElementIndex*. The name of the getter's member function is called its *ElementName*

4 A tagged getter class with DerivedCharacteristic *`D`*, ElementIndex *`N`*, and ElementName *`name`* shall provide the following interface:

```
struct __TAGGED_GETTER {
  constexpr decltype(auto) name() &       { return get<N>(static_cast<D&>(*this)); }
  constexpr decltype(auto) name() &&      { return get<N>(static_cast<D&&>(*this)); }
  constexpr decltype(auto) name() const & { return get<N>(static_cast<const D&>(*this)); }
};
```

5 A *tag specifier* is a type that facilitates a mapping from a tuple-like type and an element index into a *tagged getter* that gives named access to the element at that index. `TagSpecifier<T>` is satisfied if and only if `T` is a tag specifier. The tag specifiers in the `Tags` parameter pack shall be unique. [*Note:* The mapping mechanism from tag specifier to tagged getter is unspecified. — *end note*]

6 Let *`TAGGET`*`(D, T, `*`N`*`)` name a tagged getter type that gives named access to the *N*-th element of the tuple-like type `D`.

7 It shall not be possible to delete an instance of class template `tagged` through a pointer to any base other than `Base`.

8 `TaggedType<F>` is satisfied if and only if `F` is a unary function type with return type `T` which satisfies `TagSpecifier<T>`. Let *`TAGSPEC`*`(F)` name the tag specifier of the `TaggedType F`, and let *`TAGELEM`*`(F)` name the argument type of the `TaggedType F`.

```
tagged(Base&& that) noexcept(see below)
  requires MoveConstructible<Base>;
```

9 *Effects:* Initializes `Base` with `std::move(that)`.

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

```
is_nothrow_move_constructible_v<Base>~~::value~~
```

```
tagged(const Base& that) noexcept(see below)
  requires CopyConstructible<Base>;
```

11 *Effects:* Initializes `Base` with `that`.

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

is_nothrow_copy_constructible_v<Base>~~::value~~

```
template <class Other>
  requires Constructible<Base, Other>
constexpr tagged(tagged<Other, Tags...> &&that) noexcept(see below);
```

13 *Effects:* Initializes `Base` with `static_cast<Other&&>(that)`.

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

is_nothrow_constructible_v<Base, Other>~~::value~~

```
template <class Other>
  requires Constructible<Base, const Other&>
constexpr tagged(const tagged<Other, Tags...>& that);
```

15 *Effects:* Initializes `Base` with `static_cast<const Other&>(that)`.

```
template <class Other>
  requires Assignable<Base&, Other>
constexpr tagged& operator=(tagged<Other, Tags...>&& that) noexcept(see below);
```

16 *Effects:* Assigns `static_cast<Other&&>(that)` to `static_cast<Base&>(*this)`.

17 *Returns:* `*this`.

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

is_nothrow_assignable_v<Base&, Other>~~::value~~

```
template <class Other>
  requires Assignable<Base&, const Other&>
constexpr tagged& operator=(const tagged<Other, Tags...>& that);
```

19 *Effects:* Assigns `static_cast<const Other&>(that)` to `static_cast<Base&>(*this)`.

20 *Returns:* `*this`.

```
template <class U>
  requires Assignable<Base&, U> && !Same<decay_t<U>, tagged>
constexpr tagged& operator=(U&& u) noexcept(see below);
```

21 *Effects:* Assigns `std::forward<U>(u)` to `static_cast<Base&>(*this)`.

22 *Returns:* `*this`.

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

is_nothrow_assignable_v<Base&, U>~~::value~~

```
constexpr void swap(tagged& rhs) noexcept(see below)
  requires Swappable<Base>;
```

24 *Effects:* Calls `swap` on the result of applying `static_cast` to `*this` and `that`.

25 *Throws:* Nothing unless the call to `swap` on the `Base` sub-objects throws.

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

```
noexcept(swap(declval<Base&>(), declval<Base&>()))
```

```
friend constexpr void swap(tagged& lhs, tagged& rhs) noexcept(see below)
  requires Swappable<Base>;
```

27 *Effects:* Equivalent to `lhs.swap(rhs)`.

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

```
noexcept(lhs.swap(rhs))
```

21.2.4.3 Tuple-like access to tagged [std2.tagged.astuple]

```
namespace std {
  template <class Base, class... Tags>
  struct tuple_size<~~experimental::ranges~~::std2::tagged<Base, Tags...>>
    : tuple_size<Base> { };

  template <size_t N, class Base, class... Tags>
  struct tuple_element<N, ~~experimental::ranges~~::std2::tagged<Base, Tags...>>
    : tuple_element<N, Base> { };
}
```

21.2.4.4 Alias template tagged_pair [std2.tagged.pairs]

```
// defined in header <~~experimental/ranges~~std2/utility>

namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  // ...
  template <TaggedType T1, TaggedType T2>
  using tagged_pair = tagged<pair<TAGELEM(T1), TAGELEM(T2)>,
                             TAGSPEC(T1), TAGSPEC(T2)>;
}}~~}}~~
```

1 [*Example:*

```
// See 21.5.2:
tagged_pair<tag::min(int), tag::max(int)> p{0, 1};
assert(&p.min() == &p.first);
assert(&p.max() == &p.second);
```

— *end example*]

21.2.4.4.1 Tagged pair creation functions [std2.tagged.pairs.creation]

```
// defined in header <~~experimental/ranges~~std2/utility>

namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <TagSpecifier Tag1, TagSpecifier Tag2, class T1, class T2>
    constexpr see below make_tagged_pair(T1&& x, T2&& y);
}}~~}}~~
```

1 Let `P` be the type of `make_pair(std::forward<T1>(x), std::forward<T2>(y))`. Then the return type is `tagged<P, Tag1, Tag2>`.

2 *Returns:* `{std::forward<T1>(x), std::forward<T2>(y)}`.

3 [*Example:* In place of:

```
return tagged_pair<tag::min(int), tag::max(double)>(5, 3.1415926);   // explicit types
```

a C++ program may contain:

```
return make_tagged_pair<tag::min, tag::max>(5, 3.1415926);           // types are deduced
```

— *end example*]

21.2.4.5 Alias template tagged_tuple [std2.tagged.tuple]

1 **Header <~~experimental/ranges~~std2/tuple> synopsis**

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <TaggedType... Types>
  using tagged_tuple = tagged<tuple<TAGELEM(Types)...>,
                              TAGSPEC(Types)...>;

  template <TagSpecifier... Tags, class... Types>
    requires sizeof...(Tags) == sizeof...(Types)
      constexpr see below make_tagged_tuple(Types&&... t);
}}~~}}~~
```

2

```
template <TaggedType... Types>
using tagged_tuple = tagged<tuple<TAGELEM(Types)...>,
                            TAGSPEC(Types)...>;
```

3 [*Example:*

```
// See 21.5.2:
tagged_tuple<tag::in(char*), tag::out(char*)> t{0, 0};
assert(&t.in() == &get<0>(t));
assert(&t.out() == &get<1>(t));
```

— *end example*]

21.2.4.5.1 Tagged tuple creation functions [std2.tagged.tuple.creation]

```
template <TagSpecifier... Tags, class... Types>
  requires sizeof...(Tags) == sizeof...(Types)
    constexpr see below make_tagged_tuple(Types&&... t);
```

1 Let `T` be the type of `make_tuple(std::forward<Types>(t)...)`. Then the return type is `tagged<T, Tags...>`.

2 *Returns:* `tagged<T, Tags...>(std::forward<Types>(t)...)`.

3 [*Example:*

```
int i; float j;
make_tagged_tuple<tag::in1, tag::in2, tag::out>(1, ref(i), cref(j))
```

creates a tagged tuple of type

```
tagged_tuple<tag::in1(int), tag::in2(int&), tag::out(const float&)>
```

— *end example*]

21.3 Iterators library [std2.iterators]

21.3.1 General [std2.iterators.general]

1 This ~~C~~Subclause describes components that C++ programs may use to perform iterations over containers (~~C~~Subclause 20.8), streams (20.12.7), and stream buffers (20.12.6).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 4.

Table 4 — Iterators library summary

Subclause		Header(s)
21.3.3	Iterator requirements	
21.3.4	Indirect callable requirements	
21.3.5	Common algorithm requirements	
21.3.6	Iterator primitives	<~~experimental/ranges~~std2/iterator>
21.3.7	Predefined iterators	
21.3.8	Stream iterators	

21.3.2 Header <std2/iterator> synopsis [std2.iterator.synopsis]

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <class T> concept bool dereferenceable   // exposition only
    = requires(T& t) { {*t} -> auto&&; };

  // 21.3.3, iterator requirements:
  // 21.3.3.2, customization points:
  namespace {
    // 21.3.3.2.1, iter_move:
    inline constexpr unspecified  iter_move = unspecified ;

    // 21.3.3.2.2, iter_swap:
    inline constexpr unspecified  iter_swap = unspecified ;
  }

  // 21.3.3.3, associated types:
  // 21.3.3.3.1, difference_type:
  template <class> struct difference_type;
  template <class T> using difference_type_t
    = typename difference_type<T>::type;

  // 21.3.3.3.2, value_type:
  template <class> struct value_type;
  template <class T> using value_type_t
    = typename value_type<T>::type;

  // 21.3.3.3.3, iterator_category:
  template <class> struct iterator_category;
  template <class T> using iterator_category_t
    = typename iterator_category<T>::type;

  template <dereferenceable  T> using reference_t
    = decltype(*declval<T&>());

  template <dereferenceable  T>
      requires see below using rvalue_reference_t
    = decltype(ranges::std2::iter_move(declval<T&>()));

  // 21.3.3.4, Readable:
  template <class In>
  concept bool Readable = see below;

  // 21.3.3.5, Writable:
```

```
template <class Out, class T>
concept ~~bool~~ Writable = see below;

// 21.3.3.6, WeaklyIncrementable:
template <class I>
concept ~~bool~~ WeaklyIncrementable = see below;

// 21.3.3.7, Incrementable:
template <class I>
concept ~~bool~~ Incrementable = see below;

// 21.3.3.8, Iterator:
template <class I>
concept ~~bool~~ Iterator = see below;

// 21.3.3.9, Sentinel:
template <class S, class I>
concept ~~bool~~ Sentinel = see below;

// 21.3.3.10, SizedSentinel:
template <class S, class I>
  constexpr bool disable_sized_sentinel = false;

template <class S, class I>
concept ~~bool~~ SizedSentinel = see below;

// 21.3.3.11, InputIterator:
template <class I>
concept ~~bool~~ InputIterator = see below;

// 21.3.3.12, OutputIterator:
template <class I>
concept ~~bool~~ OutputIterator = see below;

// 21.3.3.13, ForwardIterator:
template <class I>
concept ~~bool~~ ForwardIterator = see below;

// 21.3.3.14, BidirectionalIterator:
template <class I>
concept ~~bool~~ BidirectionalIterator = see below;

// 21.3.3.15, RandomAccessIterator:
template <class I>
concept ~~bool~~ RandomAccessIterator = see below;

// 21.3.4, indirect callable requirements:
// 21.3.4.2, indirect callables:
template <class F, class I>
concept ~~bool~~ IndirectUnaryInvocable = see below;

template <class F, class I>
concept ~~bool~~ IndirectRegularUnaryInvocable = see below;

template <class F, class I>
```

```
concept ~~bool~~ IndirectUnaryPredicate = see below;

template <class F, class I1, class I2 = I1>
concept ~~bool~~ IndirectRelation = see below;

template <class F, class I1, class I2 = I1>
concept ~~bool~~ IndirectStrictWeakOrder = see below;

template <class> struct indirect_result_of;

template <class F, class... Is>
  requires Invocable<F, reference_t<Is>...>
struct indirect_result_of<F(Is...)>;

template <class F>
using indirect_result_of_t
  = typename indirect_result_of<F>::type;

// 21.3.4.3, projected:
template <Readable I, IndirectRegularUnaryInvocable<I> Proj>
struct projected;

template <WeaklyIncrementable I, class Proj>
struct difference_type<projected<I, Proj>>;

// 21.3.5, common algorithm requirements:
// 21.3.5.2 IndirectlyMovable:
template <class In, class Out>
concept ~~bool~~ IndirectlyMovable = see below;

template <class In, class Out>
concept ~~bool~~ IndirectlyMovableStorable = see below;

// 21.3.5.3 IndirectlyCopyable:
template <class In, class Out>
concept ~~bool~~ IndirectlyCopyable = see below;

template <class In, class Out>
concept ~~bool~~ IndirectlyCopyableStorable = see below;

// 21.3.5.4 IndirectlySwappable:
template <class I1, class I2 = I1>
concept ~~bool~~ IndirectlySwappable = see below;

// 21.3.5.5 IndirectlyComparable:
template <class I1, class I2, class R = equal_to<>, class P1 = identity,
    class P2 = identity>
concept ~~bool~~ IndirectlyComparable = see below;

// 21.3.5.6 Permutable:
template <class I>
concept ~~bool~~ Permutable = see below;

// 21.3.5.7 Mergeable:
template <class I1, class I2, class Out,
```

```
    class R = less<>, class P1 = identity, class P2 = identity>
concept ~~bool~~ Mergeable = see below;

template <class I, class R = less<>, class P = identity>
concept ~~bool~~ Sortable = see below;

// 21.3.6, primitives:
// 21.3.6.1, traits:
template <class Iterator> using iterator_traits = see below;

template <Readable T> using iter_common_reference_t
  = common_reference_t<reference_t<T>, value_type_t<T>&>;

// 21.3.6.3, iterator tags:
struct output_iterator_tag { };
struct input_iterator_tag { };
struct forward_iterator_tag : input_iterator_tag { };
struct bidirectional_iterator_tag : forward_iterator_tag { };
struct random_access_iterator_tag : bidirectional_iterator_tag { };

// 21.3.6.4, iterator operations:
~~namespace {~~
  inline constexpr unspecified  advance = unspecified ;
  inline constexpr unspecified  distance = unspecified ;
  inline constexpr unspecified  next = unspecified ;
  inline constexpr unspecified  prev = unspecified ;
~~}~~

// 21.3.7, predefined iterators and sentinels:

// 21.3.7.1, reverse iterators:
template <BidirectionalIterator I> class reverse_iterator;

template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator==(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator!=(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator>(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
```

```
  constexpr bool operator>=(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<=(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);

template <class I1, class I2>
    requires SizedSentinel<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
template <RandomAccessIterator I>
  constexpr reverse_iterator<I> operator+(
    difference_type_t<I> n,
    const reverse_iterator<I>& x);

template <BidirectionalIterator I>
  constexpr reverse_iterator<I> make_reverse_iterator(I i);

// 21.3.7.2, insert iterators:
template <class Container> class back_insert_iterator;
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template <class Container> class insert_iterator;
template <class Container>
  insert_iterator<Container> inserter(Container& x, iterator_t<Container> i);

// 21.3.7.3, move iterators and sentinels:
template <InputIterator I> class move_iterator;
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator==(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator!=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
```

```
  constexpr bool operator>(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator>=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);

template <class I1, class I2>
    requires SizedSentinel<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const move_iterator<I1>& x,
    const move_iterator<I2>& y);
template <RandomAccessIterator I>
  constexpr move_iterator<I> operator+(
    difference_type_t<I> n,
    const move_iterator<I>& x);
template <InputIterator I>
  constexpr move_iterator<I> make_move_iterator(I i);

template <Semiregular S> class move_sentinel;

template <class I, Sentinel<I> S>
  constexpr bool operator==(
    const move_iterator<I>& i, const move_sentinel<S>& s);
template <class I, Sentinel<I> S>
  constexpr bool operator==(
    const move_sentinel<S>& s, const move_iterator<I>& i);
template <class I, Sentinel<I> S>
  constexpr bool operator!=(
    const move_iterator<I>& i, const move_sentinel<S>& s);
template <class I, Sentinel<I> S>
  constexpr bool operator!=(
    const move_sentinel<S>& s, const move_iterator<I>& i);

template <class I, SizedSentinel<I> S>
  constexpr difference_type_t<I> operator-(
    const move_sentinel<S>& s, const move_iterator<I>& i);
template <class I, SizedSentinel<I> S>
  constexpr difference_type_t<I> operator-(
    const move_iterator<I>& i, const move_sentinel<S>& s);

template <Semiregular S>
  constexpr move_sentinel<S> make_move_sentinel(S s);

// 21.3.7.4, common iterators:
template <Iterator I, Sentinel<I> S>
  requires !Same<I, S>
class common_iterator;

template <Readable I, class S>
struct value_type<common_iterator<I, S>>;

template <InputIterator I, class S>
struct iterator_category<common_iterator<I, S>>;
```

```
template <ForwardIterator I, class S>
struct iterator_category<common_iterator<I, S>>;

template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator==(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  requires EqualityComparableWith<I1, I2>
bool operator==(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator!=(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

template <class I2, SizedSentinel<I2> I1, SizedSentinel<I2> S1, SizedSentinel<I1> S2>
difference_type_t<I2> operator-(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

// 21.3.7.5, default sentinels:
class default_sentinel;

// 21.3.7.6, counted iterators:
template <Iterator I> class counted_iterator;

template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I>
  constexpr bool operator==(
    const counted_iterator<~~auto~~I>& x, default_sentinel);
template <class I>
  constexpr bool operator==(
    default_sentinel, const counted_iterator<~~auto~~I>& x);
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I>
  constexpr bool operator!=(
    const counted_iterator<~~auto~~I>& x, default_sentinel y);
template <class I>
  constexpr bool operator!=(
    default_sentinel x, const counted_iterator<~~auto~~I>& y);
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>(
```

```
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I>
  constexpr difference_type_t<I> operator-(
    const counted_iterator<I>& x, default_sentinel y);
template <class I>
  constexpr difference_type_t<I> operator-(
    default_sentinel x, const counted_iterator<I>& y);
template <RandomAccessIterator I>
  constexpr counted_iterator<I>
    operator+(difference_type_t<I> n, const counted_iterator<I>& x);
template <Iterator I>
  constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);

// 21.3.7.7, unreachable sentinels:
class unreachable;
template <Iterator I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator==(unreachable, const I&) noexcept;
template <Iterator I>
  constexpr bool operator!=(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator!=(unreachable, const I&) noexcept;

// 21.4.7, dangling wrapper:
template <class T> class dangling;

// 21.3.8, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(default_sentinel x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
          default_sentinel y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
 bool operator!=(default_sentinel x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
```

```
    bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
            default_sentinel y);

  template <class T, class charT = char, class traits = char_traits<charT>>
      class ostream_iterator;

  template <class charT, class traits = char_traits<charT> >
    class istreambuf_iterator;
  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT, traits>& a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator==(default_sentinel a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT, traits>& a,
            default_sentinel b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT, traits>& a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator!=(default_sentinel a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT, traits>& a,
            default_sentinel b);

  template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator;
}}~~}}~~

namespace std {
  // 21.3.6.2, iterator traits:
  template <~~experimental::ranges~~::std2::Iterator Out>
    struct iterator_traits<Out>;
  template <~~experimental::ranges~~::std2::InputIterator In>
    struct iterator_traits<In>;
  template <~~experimental::ranges~~::std2::InputIterator In>
      requires ~~experimental::ranges~~::std2::Sentinel<In, In>
    struct iterator_traits;
}
```

21.3.3 Iterator requirements [std2.iterator.requirements]

21.3.3.1 General [std2.iterator.requirements.general]

1 Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. All input iterators `i` support the expression `*i`, resulting in a value of some object type `T`, called the *value type* of the iterator. All output iterators support the expression `*i = o` where `o` is a value of some type that is in the set of types that are *writable* to the particular iterator type of `i`. For every iterator type `X` there is a corresponding signed integer type called the *difference type* of the iterator.

2 Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics

of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This document defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 5.

Table 5 — Relations among iterator categories

Random Access	→ **Bidirectional**	→ **Forward**	→ **Input**
			→ **Output**

3 The five categories of iterators correspond to the iterator concepts `InputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator`, and `RandomAccessIterator`, respectively. The generic term *iterator* refers to any type that satisfies `Iterator`.

4 Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

5 Iterators that further satisfy the requirements of output iterators are called *mutable iterator*s. Nonmutable iterators are referred to as *constant iterator*s.

6 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [*Example:* After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a singular value of a pointer. — *end example*] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and using a value-initialized iterator as the source of a copy or move operation. [*Note:* This guarantee is not offered for default initialization, although the distinction only matters for types with trivial default constructors such as pointers or aggregates holding pointers. — *end note*] In these cases the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

7 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a *sentinel* that designate the beginning and end of the computation, or an iterator and a count that designate the beginning and the number of elements to which the computation is to be applied.

8 An iterator and a sentinel denoting a range are comparable. The types of a sentinel and an iterator that denote a range must satisfy `Sentinel` (21.3.3.9). A range `[i,s)` is empty if `i == s`; otherwise, `[i,s)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the first iterator `j` such that `j == s`.

9 A sentinel `s` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == s`. If `s` is reachable from `i`, `[i,s)` denotes a range.

10 A counted range `[i,n)` is empty if `n == 0`; otherwise, `[i,n)` refers to the `n` elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the result of incrementing `i` `n` times.

11 A range `[i,s)` is valid if and only if `s` is reachable from `i`. A counted range `[i,n)` is valid if and only if `n == 0`; or `n` is positive, `i` is dereferenceable, and `[++i,--n)` is valid. The result of the application of functions in the library to invalid ranges is undefined.

12 All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized).

13 Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

14 An *invalid* iterator is an iterator that may be singular.[3]

21.3.3.2 Customization points [std2.iterator.custpoints]

21.3.3.2.1 `iter_move` [std2.iterator.custpoints.iter_move]

1 The name `iter_move` denotes a *customization point object* (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::iter_move(E)` for some subexpression `E` is expression-equivalent to the following:

(1.1) — `static_cast<decltype(iter_move(E))>(iter_move(E))`, if that expression is well-formed when evaluated in a context that does not include ~~`ranges`~~`::std2::iter_move` but does include the lookup set produced by argument-dependent lookup (6.4.2).

(1.2) — Otherwise, if the expression `*E` is well-formed:

(1.2.1) — if `*E` is an lvalue, `std::move(*E)`;

(1.2.2) — otherwise, `static_cast<decltype(*E)>(*E)`.

(1.3) — Otherwise, ~~`ranges`~~`::std2::iter_move(E)` is ill-formed.

2 If ~~`ranges`~~`::std2::iter_move(E)` does not equal `*E`, the program is ill-formed with no diagnostic required.

21.3.3.2.2 `iter_swap` [std2.iterator.custpoints.iter_swap]

1 The name `iter_swap` denotes a *customization point object* (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::iter_swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to the following:

(1.1) — `(void)iter_swap(E1, E2)`, if that expression is well-formed when evaluated in a context that does not include ~~`ranges`~~`::std2::iter_swap` but does include the lookup set produced by argument-dependent lookup (6.4.2) and the following declaration:

`template <class I1, class I2>`
`void iter_swap(`~~`auto, auto`~~`I1, I2) = delete;`

(1.2) — Otherwise, if the types of `E1` and `E2` both satisfy `Readable`, and if the reference type of `E1` is swappable with (20.3.3.11) the reference type of `E2`, then ~~`ranges`~~`::std2::swap(*E1, *E2)`

(1.3) — Otherwise, if the types `T1` and `T2` of `E1` and `E2` satisfy `IndirectlyMovableStorable<T1, T2> && IndirectlyMovableStorable<T2, T1>`, `(void)(*E1 = iter_exchange_move(E2, E1))`, except that `E1` is evaluated only once.

(1.4) — Otherwise, ~~`ranges`~~`::std2::iter_swap(E1, E2)` is ill-formed.

2 If ~~`ranges`~~`::std2::iter_swap(E1, E2)` does not swap the values denoted by the expressions `E1` and `E2`, the program is ill-formed with no diagnostic required.

3 `iter_exchange_move` is an exposition-only function specified as:

```
template <class X, class Y>
  constexpr value_type_t<remove_reference_t<X>> iter_exchange_move(X&& x, Y&& y)
    noexcept(see below);
```

4 *Effects:* Equivalent to:

3) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

```
value_type_t<remove_reference_t<X>> old_value(iter_move(x));
*x = iter_move(y);
return old_value;
```

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

```
NE(remove_reference_t<X>, remove_reference_t<Y>) &&
NE(remove_reference_t<Y>, remove_reference_t<X>)
```

Where `NE(T1, T2)` is the expression:

```
is_nothrow_constructible_v<value_type_t<T1>, rvalue_reference_t<T1>>~~::value~~ &&
is_nothrow_assignable_v<value_type_t<T1>&, rvalue_reference_t<T1>>~~::value~~ &&
is_nothrow_assignable_v<reference_t<T1>, rvalue_reference_t<T2>>~~::value~~ &&
is_nothrow_assignable_v<reference_t<T1>, value_type_t<T2>>~~::value~~> &&
is_nothrow_move_constructible_v<value_type_t<T1>>~~::value~~ &&
noexcept(~~ranges~~::std2::iter_move(declval<T1&>()))
```

21.3.3.3 Iterator associated types [std2.iterator.assoc.types]

1 To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if `WI` is the name of a type that satisfies the `WeaklyIncrementable` concept (21.3.3.6), `R` is the name of a type that satisfies the `Readable` concept (21.3.3.4), and `II` is the name of a type that satisfies the `InputIterator` concept (21.3.3.11) concept, the types

```
difference_type_t<WI>
value_type_t<R>
iterator_category_t<II>
```

be defined as the iterator's difference type, value type and iterator category, respectively.

21.3.3.3.1 `difference_type` [std2.iterator.assoc.types.difference_type]

1 `difference_type_t<T>` is implemented as if:

```
template <class> struct difference_type { };

template <class T>
struct difference_type<T*>
  : enable_if<is_object_v<T>~~::value~~, ptrdiff_t> { };

template <class I>
struct difference_type<const I> : difference_type<decay_t<I>> { };

template <class T>
  requires requires { typename T::difference_type; }
struct difference_type<T> {
  using type = typename T::difference_type;
};

template <class T>
  requires !requires { typename T::difference_type; } &&
    requires(const T& a, const T& b) { { a - b } -> Integral; }
struct difference_type<T>
  : make_signed< decltype(declval<T>() - declval<T>()) > {
};
```

```
template <class T> using difference_type_t
  = typename difference_type<T>::type;
```

2 Users may specialize `difference_type` on user-defined types.

21.3.3.3.2 `value_type` [std2.iterator.assoc.types.value_type]

1 A `Readable` type has an associated value type that can be accessed with the `value_type_t` alias template.

```
template <class> struct value_type { };

template <class T>
struct value_type<T*>
  : enable_if<is_object<T>~~::value~~, remove_cv_t<T>> { };

template <class I>
  requires is_array_v<I>~~::value~~
struct value_type<I> : value_type<decay_t<I>> { };

template <class I>
struct value_type<const I> : value_type<decay_t<I>> { };

template <class T>
  requires requires { typename T::value_type; }
struct value_type<T>
  : enable_if<is_object_v<typename T::value_type>~~::value~~, typename T::value_type> { };

template <class T>
  requires requires { typename T::element_type; }
struct value_type<T>
  : enable_if<
      is_object_v<typename T::element_type>~~::value~~,
      remove_cv_t<typename T::element_type>>
  { };

template <class T> using value_type_t
  = typename value_type<T>::type;
```

2 If a type `I` has an associated value type, then `value_type<I>::type` shall name the value type. Otherwise, there shall be no nested type `type`.

3 The `value_type` class template may be specialized on user-defined types.

4 When instantiated with a type `I` such that `I::value_type` is valid and denotes a type, `value_type<I>::type` names that type, unless it is not an object type (6.7) in which case `value_type<I>` shall have no nested type `type`. [*Note:* Some legacy output iterators define a nested type named `value_type` that is an alias for `void`. These types are not `Readable` and have no associated value types. — *end note*]

5 When instantiated with a type `I` such that `I::element_type` is valid and denotes a type, `value_type<I>::type` names the type `remove_cv_t<I::element_type>`, unless it is not an object type (6.7) in which case `value_type<I>` shall have no nested type `type`. [*Note:* Smart pointers like `shared_ptr<int>` are `Readable` and have an associated value type. But a smart pointer like `shared_ptr<void>` is not `Readable` and has no associated value type. — *end note*]

21.3.3.3.3 `iterator_category` [std2.iterator.assoc.types.iterator_category]

1 `iterator_category_t<T>` is implemented as if:

```
template <class> struct iterator_category { };
```

```
template <class T>
struct iterator_category<T*>
  : enable_if<is_object_v<T>~~::value~~, random_access_iterator_tag> { };

template <class T>
struct iterator_category<T const> : iterator_category<T> { };

template <class T>
  requires requires { typename T::iterator_category; }
struct iterator_category<T> {
  using type = see below;
};

template <class T> using iterator_category_t
  = typename iterator_category<T>::type;
```

2 Users may specialize `iterator_category` on user-defined types.

3 If `T::iterator_category` is valid and denotes a type, then the type `iterator_category<T>::type` is computed as follows:

(3.1) — If `T::iterator_category` is the same as or derives from `std::random_access_iterator_tag`, `iterator_category<T>::type` is ~~`ranges`~~`::std2::random_access_iterator_tag`.

(3.2) — Otherwise, if `T::iterator_category` is the same as or derives from `std::bidirectional_iterator_tag`, `iterator_category<T>::type` is ~~`ranges`~~`::std2::bidirectional_iterator_tag`.

(3.3) — Otherwise, if `T::iterator_category` is the same as or derives from `std::forward_iterator_tag`, `iterator_category<T>::type` is ~~`ranges`~~`::std2::forward_iterator_tag`.

(3.4) — Otherwise, if `T::iterator_category` is the same as or derives from `std::input_iterator_tag`, `iterator_category<T>::type` is ~~`ranges`~~`::std2::input_iterator_tag`.

(3.5) — Otherwise, if `T::iterator_category` is the same as or derives from `std::output_iterator_tag`, `iterator_category<T>` has no nested `type`.

(3.6) — Otherwise, `iterator_category<T>::type` is `T::iterator_category`

4 `rvalue_reference_t<T>` is implemented as if:

```
template <dereferenceable  T>
    requires see below using rvalue_reference_t
  = decltype(~~ranges~~::std2::iter_move(declval<T&>()));
```

5 The expression in the `requires` clause is equivalent to:

```
requires(T& t) { { ~~ranges~~::std2::iter_move(t) } -> auto&&; }
```

21.3.3.4 Concept `Readable` [std2.iterators.readable]

1 The `Readable` concept is satisfied by types that are readable by applying `operator*` including pointers, smart pointers, and iterators.

```
template <class In>
concept ~~bool~~ Readable =
  requires {
```

```
    typename value_type_t<In>;
    typename reference_t<In>;
    typename rvalue_reference_t<In>;
  } &&
  CommonReference<reference_t<In>&&, value_type_t<In>&> &&
  CommonReference<reference_t<In>&&, rvalue_reference_t<In>&&> &&
  CommonReference<rvalue_reference_t<In>&&, const value_type_t<In>&>;
```

21.3.3.5 Concept `Writable` [std2.iterators.writable]

1 The `Writable` concept specifies the requirements for writing a value into an iterator's referenced object.

```
template <class Out, class T>
concept bool Writable =
  requires(Out&& o, T&& t) {
    *o = std::forward<T>(t); // not required to be equality preserving
    *std::forward<Out>(o) = std::forward<T>(t); // not required to be equality preserving
    const_cast<const reference_t<Out>&&>(*o) =
      std::forward<T>(t); // not required to be equality preserving
    const_cast<const reference_t<Out>&&>(*std::forward<Out>(o)) =
      std::forward<T>(t); // not required to be equality preserving
  };
```

2 Let `E` be an an expression such that `decltype((E))` is `T`, and let `o` be a dereferenceable object of type `Out`. `Writable<Out, T>` is satisfied only if

(2.1) — If `Readable<Out> && Same<value_type_t<Out>, decay_t<T>>` is satisfied, then `*o` after any above assignment is equal to the value of `E` before the assignment.

3 After evaluating any above assignment expression, `o` is not required to be dereferenceable.

4 If `E` is an xvalue (), the resulting state of the object it denotes is valid but unspecified ().

5 [*Note:* The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the writable type happens only once.* — *end note*]

21.3.3.6 Concept `WeaklyIncrementable` [std2.iterators.weaklyincrementable]

1 The `WeaklyIncrementable` concept specifies the requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are not required to be equality-preserving, nor is the type required to be `EqualityComparable`.

```
template <class I>
concept bool WeaklyIncrementable =
  Semiregular<I> &&
  requires(I i) {
    typename difference_type_t<I>;
    requires SignedIntegral<difference_type_t<I>>;
    { ++i } -> Same<I>&; // not required to be equality preserving
    i++; // not required to be equality preserving
  };
```

2 Let `i` be an object of type `I`. When `i` is in the domain of both pre- and post-increment, `i` is said to be *incrementable*. `WeaklyIncrementable<I>` is satisfied only if

(2.1) — The expressions `++i` and `i++` have the same domain.

(2.2) — If `i` is incrementable, then both `++i` and `i++` advance `i` to the next element.

(2.3) — If `i` is incrementable, then `&++i` is equal to `&i`.

3 [*Note:* For `WeaklyIncrementable` types, `a` equals `b` does not imply that `++a` equals `++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on weakly incrementable types should never attempt to pass through the same incrementable value twice. They should be single pass algorithms. These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

21.3.3.7 Concept `Incrementable` [std2.iterators.incrementable]

1 The `Incrementable` concept specifies requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be `EqualityComparable`. [*Note:* This requirement supersedes the annotations on the increment expressions in the definition of `WeaklyIncrementable`. — *end note*]

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

2 Let `a` and `b` be incrementable objects of type `I`. `Incrementable<I>` is satisfied only if

(2.1) — If `bool(a == b)` then `bool(a++ == b)`.

(2.2) — If `bool(a == b)` then `bool((a++, a) == ++b)`.

3 [*Note:* The requirement that `a` equals `b` implies `++a` equals `++b` (which is not true for weakly incrementable types) allows the use of multi-pass one-directional algorithms with types that satisfy `Incrementable`. — *end note*]

21.3.3.8 Concept `Iterator` [std2.iterators.iterator]

1 The `Iterator` concept forms the basis of the iterator concept taxonomy; every iterator satisfies the `Iterator` requirements. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (21.3.3.9), to read (21.3.3.11) or write (21.3.3.12) values, or to provide a richer set of iterator movements (21.3.3.13, 21.3.3.14, 21.3.3.15).)

```
template <class I>
concept ~~bool~~ Iterator =
  requires(I i) {
    { *i } -> auto&&; // Requires: i is dereferenceable
  } &&
  WeaklyIncrementable<I>;
```

2 [*Note:* The requirement that the result of dereferencing the iterator is deducible from `auto&&` means that it cannot be `void`. — *end note*]

21.3.3.9 Concept `Sentinel` [std2.iterators.sentinel]

1 The `Sentinel` concept specifies the relationship between an `Iterator` type and a `Semiregular` type whose values denote a range.

```
template <class S, class I>
concept ~~bool~~ Sentinel =
  Semiregular<S> &&
  Iterator<I> &&
  WeaklyEqualityComparableWith<S, I>;
```

2 Let `s` and `i` be values of type `S` and `I` such that `[i,s)` denotes a range. Types `S` and `I` satisfy `Sentinel<S, I>` only if:

(2.1) — `i == s` is well-defined.

(2.2) — If `bool(i != s)` then `i` is dereferenceable and `[++i,s)` denotes a range.

3 The domain of `==` can change over time. Given an iterator `i` and sentinel `s` such that `[i,s)` denotes a range and `i != s`, `[i,s)` is not required to continue to denote a range after incrementing any iterator equal to `i`. Consequently, `i == s` is no longer required to be well-defined.

21.3.3.10 Concept `SizedSentinel` [std2.iterators.sizedsentinel]

1 The `SizedSentinel` concept specifies requirements on an `Iterator` and a `Sentinel` that allow the use of the `-` operator to compute the distance between them in constant time.

```
template <class S, class I>
concept ~~bool~~ SizedSentinel =
  Sentinel<S, I> &&
  !disable_sized_sentinel<remove_cv_t<S>, remove_cv_t<I>> &&
  requires(const I& i, const S& s) {
    { s - i } -> Same<difference_type_t<I>>&&;
    { i - s } -> Same<difference_type_t<I>>&&;
  };
```

2 Let `i` be an iterator of type `I`, and `s` a sentinel of type `S` such that `[i,s)` denotes a range. Let N be the smallest number of applications of `++i` necessary to make `bool(i == s)` be `true`. `SizedSentinel<S, I>` is satisfied only if:

(2.1) — If N is representable by `difference_type_t<I>`, then `s - i` is well-defined and equals N.

(2.2) — If $-N$ is representable by `difference_type_t<I>`, then `i - s` is well-defined and equals $-N$.

3 [*Note:* `disable_sized_sentinel` provides a mechanism to enable use of sentinels and iterators with the library that meet the syntactic requirements but do not in fact satisfy `SizedSentinel`. A program that instantiates a library template that requires `SizedSentinel` with an iterator type `I` and sentinel type `S` that meet the syntactic requirements of `SizedSentinel<S, I>` but do not satisfy `SizedSentinel` is ill-formed with no diagnostic required unless `disable_sized_sentinel<S, I>` evaluates to `true` (21.1.2.1.3). — *end note*]

4 [*Note:* The `SizedSentinel` concept is satisfied by pairs of `RandomAccessIterators` (21.3.3.15) and by counted iterators and their sentinels (21.3.7.6.1). — *end note*]

21.3.3.11 Concept `InputIterator` [std2.iterators.input]

1 The `InputIterator` concept is a refinement of `Iterator` (21.3.3.8). It defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (21.3.3.4)) and which can be both pre- and post-incremented. [*Note:* Unlike in ISO/IEC 14882, input iterators are not required to satisfy `EqualityComparable` (20.3.4.3). — *end note*]

```
template <class I>
concept ~~bool~~ InputIterator =
  Iterator<I> &&
  Readable<I> &&
  requires { typename iterator_category_t<I>; } &&
  DerivedFrom<iterator_category_t<I>, input_iterator_tag>;
```

21.3.3.12 Concept `OutputIterator` [std2.iterators.output]

1 The `OutputIterator` concept is a refinement of `Iterator` (21.3.3.8). It defines requirements for a type that can be used to write values (from the requirement for `Writable` (21.3.3.5)) and which can be both pre- and post-incremented. However, output iterators are not required to satisfy `EqualityComparable`.

```
template <class I, class T>
concept ~~bool~~ OutputIterator =
  Iterator<I> &&
  Writable<I, T> &&
  requires(I i, T&& t) {
    *i++ = std::forward<T>(t); // not required to be equality preserving
  };
```

2 Let `E` be an expression such that `decltype((E))` is `T`, and let `i` be a dereferenceable object of type `I`. `OutputIterator<I, T>` is satisfied only if `*i++ = E;` has effects equivalent to:

```
*i = E;
++i;
```

3 [*Note:* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — *end note*]

21.3.3.13 Concept `ForwardIterator` [std2.iterators.forward]

1 The `ForwardIterator` concept refines `InputIterator` (21.3.3.11), adding equality comparison and the multi-pass guarantee, specified below.

```
template <class I>
concept ~~bool~~ ForwardIterator =
  InputIterator<I> &&
  DerivedFrom<iterator_category_t<I>, forward_iterator_tag> &&
  Incrementable<I> &&
  Sentinel<I, I>;
```

2 The domain of `==` for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators of the same type may be compared and shall compare equal to other value-initialized iterators of the same type. [*Note:* Value-initialized iterators behave as if they refer past the end of the same empty sequence. — *end note*]

3 Pointers and references obtained from a forward iterator into a range `[i,s)` shall remain valid while `[i,s)` continues to denote a range.

4 Two dereferenceable iterators `a` and `b` of type `X` offer the *multi-pass guarantee* if:

(4.1) — `a == b` implies `++a == ++b` and

(4.2) — The expression `([](X x){++x;}(a), *a)` is equivalent to the expression `*a`.

5 [*Note:* The requirement that `a == b` implies `++a == ++b` (which is not true for weaker iterators) and the removal of the restrictions on the number of assignments through a mutable iterator (which applies to output iterators) allow the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

21.3.3.14 Concept `BidirectionalIterator` [std2.iterators.bidirectional]

1 The `BidirectionalIterator` concept refines `ForwardIterator` (21.3.3.13), and adds the ability to move an iterator backward as well as forward.

```
template <class I>
concept ~~bool~~ BidirectionalIterator =
  ForwardIterator<I> &&
  DerivedFrom<iterator_category_t<I>, bidirectional_iterator_tag> &&
  requires(I i) {
    { --i } -> Same<I>&;
    { i-- } -> Same<I>&&;
  };
```

2 A bidirectional iterator `r` is decrementable if and only if there exists some `s` such that `++s == r`. Decrementable iterators `r` shall be in the domain of the expressions `--r` and `r--`.

3 Let `a` and `b` be decrementable objects of type `I`. `BidirectionalIterator<I>` is satisfied only if:

(3.1) — `&--a == &a`.

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

(3.3) — If `bool(a == b)`, then after evaluating both `a--` and `--b`, `bool(a == b)` still holds.

(3.4) — If `a` is incrementable and `bool(a == b)`, then `bool(--(++a) == b)`.

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

21.3.3.15 Concept `RandomAccessIterator` [std2.iterators.random.access]

1 The `RandomAccessIterator` concept refines `BidirectionalIterator` (21.3.3.14) and adds support for constant-time advancement with `+=`, `+`, `-=`, and `-`, and the computation of distance in constant time with `-`. Random access iterators also support array notation via subscripting.

```
template <class I>
concept ~~bool~~ RandomAccessIterator =
  BidirectionalIterator<I> &&
  DerivedFrom<iterator_category_t<I>, random_access_iterator_tag> &&
  StrictTotallyOrdered<I> &&
  SizedSentinel<I, I> &&
  requires(I i, const I j, const difference_type_t<I> n) {
    { i += n } -> Same<I>&;
    { j + n }  -> Same<I>&&;
    { n + j }  -> Same<I>&&;
    { i -= n } -> Same<I>&;
    { j - n }  -> Same<I>&&;
    j[n];
    requires Same<decltype(j[n]), reference_t<I>>;
  };
```

2 Let `a` and `b` be valid iterators of type `I` such that `b` is reachable from `a`. Let `n` be the smallest value of type `difference_type_t<I>` such that after `n` applications of `++a`, then `bool(a == b)`. `RandomAccessIterator<I>` is satisfied only if:

(2.1) — `(a += n)` is equal to `b`.

(2.2) — `&(a += n)` is equal to `&a`.

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

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

(2.5) — `a + 0` is equal to `a`.

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

(2.7) — `(b += -n)` is equal to `a`.

(2.8) — `(b -= n)` is equal to `a`.

(2.9) — `&(b -= n)` is equal to `&b`.

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

(2.11) — If `b` is dereferenceable, then `a[n]` is valid and is equal to `*b`.

21.3.4 Indirect callable requirements [std2.indirectcallable]

21.3.4.1 General [std2.indirectcallable.general]

1 There are several concepts that group requirements of algorithms that take callable objects (20.5.14.3) as arguments.

21.3.4.2 Indirect callables [std2.indirectcallable.indirectinvocable]

1 The indirect callable concepts are used to constrain those algorithms that accept callable objects (20.5.14.2) as arguments.

```
template <class F, class I>
concept ~~bool~~ IndirectUnaryInvocable =
  Readable<I> &&
  CopyConstructible<F> &&
  Invocable<F&, value_type_t<I>&> &&
  Invocable<F&, reference_t<I>> &&
  Invocable<F&, iter_common_reference_t<I>> &&
  CommonReference<
    result_of_t<F&(value_type_t<I>&)>,
    result_of_t<F&(reference_t<I>&&)>>;

template <class F, class I>
concept ~~bool~~ IndirectRegularUnaryInvocable =
  Readable<I> &&
  CopyConstructible<F> &&
  RegularInvocable<F&, value_type_t<I>&> &&
  RegularInvocable<F&, reference_t<I>> &&
  RegularInvocable<F&, iter_common_reference_t<I>> &&
  CommonReference<
    result_of_t<F&(value_type_t<I>&)>,
    result_of_t<F&(reference_t<I>&&)>>;

template <class F, class I>
concept ~~bool~~ IndirectUnaryPredicate =
  Readable<I> &&
  CopyConstructible<F> &&
  Predicate<F&, value_type_t<I>&> &&
  Predicate<F&, reference_t<I>> &&
  Predicate<F&, iter_common_reference_t<I>>;

template <class F, class I1, class I2 = I1>
concept ~~bool~~ IndirectRelation =
  Readable<I1> && Readable<I2> &&
```

```
    CopyConstructible<F> &&
    Relation<F&, value_type_t<I1>&, value_type_t<I2>&> &&
    Relation<F&, value_type_t<I1>&, reference_t<I2>> &&
    Relation<F&, reference_t<I1>, value_type_t<I2>&> &&
    Relation<F&, reference_t<I1>, reference_t<I2>> &&
    Relation<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

  template <class F, class I1, class I2 = I1>
  concept ~~bool~~ IndirectStrictWeakOrder =
    Readable<I1> && Readable<I2> &&
    CopyConstructible<F> &&
    StrictWeakOrder<F&, value_type_t<I1>&, value_type_t<I2>&> &&
    StrictWeakOrder<F&, value_type_t<I1>&, reference_t<I2>> &&
    StrictWeakOrder<F&, reference_t<I1>, value_type_t<I2>&> &&
    StrictWeakOrder<F&, reference_t<I1>, reference_t<I2>> &&
    StrictWeakOrder<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

  template <class> struct indirect_result_of { };

  template <class F, class... Is>
    requires Invocable<F, reference_t<Is>...>
  struct indirect_result_of<F(Is...)> :
    result_of<F(reference_t<Is>&&...)> { };
```

21.3.4.3 Class template `projected` [std2.projected]

1 The `projected` class template is intended for use when specifying the constraints of algorithms that accept callable objects and projections (3.1). It bundles a `Readable` type `I` and a function `Proj` into a new `Readable` type whose `reference` type is the result of applying `Proj` to the `reference_t` of `I`.

```
template <Readable I, IndirectRegularUnaryInvocable<I> Proj>
struct projected {
  using value_type = remove_cv_t<remove_reference_t<indirect_result_of_t<Proj&(I)>>>;
  indirect_result_of_t<Proj&(I)> operator*() const;
};

template <WeaklyIncrementable I, class Proj>
struct difference_type<projected<I, Proj>> {
  using type = difference_type_t<I>;
};
```

2 [*Note:* `projected` is only used to ease constraints specification. Its member function need not be defined. — *end note*]

21.3.5 Common algorithm requirements [std2.commonalgoreq]

21.3.5.1 General [std2.commonalgoreq.general]

1 There are several additional iterator concepts that are commonly applied to families of algorithms. These group together iterator requirements of algorithm families. There are three relational concepts that specify how element values are transferred between `Readable` and `Writable` types: `IndirectlyMovable`, `IndirectlyCopyable`, and `IndirectlySwappable`. There are three relational concepts for rearrangements: `Permutable`, `Mergeable`, and `Sortable`. There is one relational concept for comparing values from different sequences: `IndirectlyComparable`.

2 [*Note:* The `equal_to<>` and `less<>` (21.2.3.1) function types used in the concepts below impose additional constraints on their arguments beyond those that appear explicitly in the concepts' bodies. `equal_to<>`

requires its arguments satisfy `EqualityComparableWith` (20.3.4.3), and `less<>` requires its arguments satisfy `StrictTotallyOrderedWith` (20.3.4.4). — *end note*]

21.3.5.2 Concept IndirectlyMovable [std2.commonalgoreq.indirectlymovable]

1 The `IndirectlyMovable` concept specifies the relationship between a `Readable` type and a `Writable` type between which values may be moved.

```
template <class In, class Out>
concept bool IndirectlyMovable =
  Readable<In> &&
  Writable<Out, rvalue_reference_t<In>>;
```

2 The `IndirectlyMovableStorable` concept augments `IndirectlyMovable` with additional requirements enabling the transfer to be performed through an intermediate object of the `Readable` type's value type.

```
template <class In, class Out>
concept bool IndirectlyMovableStorable =
  IndirectlyMovable<In, Out> &&
  Writable<Out, value_type_t<In>> &&
  Movable<value_type_t<In>> &&
  Constructible<value_type_t<In>, rvalue_reference_t<In>> &&
  Assignable<value_type_t<In>&, rvalue_reference_t<In>>;
```

21.3.5.3 Concept IndirectlyCopyable [std2.commonalgoreq.indirectlycopyable]

1 The `IndirectlyCopyable` concept specifies the relationship between a `Readable` type and a `Writable` type between which values may be copied.

```
template <class In, class Out>
concept bool IndirectlyCopyable =
  Readable<In> &&
  Writable<Out, reference_t<In>>;
```

2 The `IndirectlyCopyableStorable` concept augments `IndirectlyCopyable` with additional requirements enabling the transfer to be performed through an intermediate object of the `Readable` type's value type. It also requires the capability to make copies of values.

```
template <class In, class Out>
concept bool IndirectlyCopyableStorable =
  IndirectlyCopyable<In, Out> &&
  Writable<Out, const value_type_t<In>&> &&
  Copyable<value_type_t<In>> &&
  Constructible<value_type_t<In>, reference_t<In>> &&
  Assignable<value_type_t<In>&, reference_t<In>>;
```

21.3.5.4 Concept IndirectlySwappable [std2.commonalgoreq.indirectlyswappable]

1 The `IndirectlySwappable` concept specifies a swappable relationship between the values referenced by two `Readable` types.

```
template <class I1, class I2 = I1>
concept bool IndirectlySwappable =
  Readable<I1> && Readable<I2> &&
  requires(I1&& i1, I2&& i2) {
    ranges::std2::iter_swap(std::forward<I1>(i1), std::forward<I2>(i2));
    ranges::std2::iter_swap(std::forward<I2>(i2), std::forward<I1>(i1));
    ranges::std2::iter_swap(std::forward<I1>(i1), std::forward<I1>(i1));
```

```
    ~~ranges~~::std2::iter_swap(std::forward<I2>(i2), std::forward<I2>(i2));
  };
```

2 Given an object `i1` of type `I1` and an object `i2` of type `I2`, `IndirectlySwappable<I1, I2>` is satisfied if after ~~ranges~~::std2::iter_swap(i1, i2), the value of `*i1` is equal to the value of `*i2` before the call, and *vice versa*.

21.3.5.5 Concept IndirectlyComparable [std2.commonalgoreq.indirectlycomparable]

1 The `IndirectlyComparable` concept specifies the common requirements of algorithms that compare values from two different sequences.

```
template <class I1, class I2, class R = equal_to<>, class P1 = identity,
  class P2 = identity>
concept ~~bool~~ IndirectlyComparable =
  IndirectRelation<R, projected<I1, P1>, projected<I2, P2>>;
```

21.3.5.6 Concept Permutable [std2.commonalgoreq.permutable]

1 The `Permutable` concept specifies the common requirements of algorithms that reorder elements in place by moving or swapping them.

```
template <class I>
concept ~~bool~~ Permutable =
  ForwardIterator<I> &&
  IndirectlyMovableStorable<I, I> &&
  IndirectlySwappable<I, I>;
```

21.3.5.7 Concept Mergeable [std2.commonalgoreq.mergeable]

1 The `Mergeable` concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by copying elements.

```
template <class I1, class I2, class Out,
    class R = less<>, class P1 = identity, class P2 = identity>
concept ~~bool~~ Mergeable =
  InputIterator<I1> &&
  InputIterator<I2> &&
  WeaklyIncrementable<Out> &&
  IndirectlyCopyable<I1, Out> &&
  IndirectlyCopyable<I2, Out> &&
  IndirectStrictWeakOrder<R, projected<I1, P1>, projected<I2, P2>>;
```

21.3.5.8 Concept Sortable [std2.commonalgoreq.sortable]

1 The `Sortable` concept specifies the common requirements of algorithms that permute sequences into ordered sequences (e.g., `sort`).

```
template <class I, class R = less<>, class P = identity>
concept ~~bool~~ Sortable =
  Permutable<I> &&
  IndirectStrictWeakOrder<R, projected<I, P>>;
```

21.3.6 Iterator primitives [std2.iterator.primitives]

1 To simplify the task of defining iterators, the library provides several classes and functions:

21.3.6.1 Iterator traits [std2.iterator.traits]

1 For the sake of backwards compatibility, this document specifies the existence of an `iterator_traits` alias that collects an iterator's associated types. It is defined as if:

```
template <InputIterator I> struct __pointer_type {        // exposition only
  using type = add_pointer_t<reference_t<I>>;
};
template <InputIterator I>
  requires requires(I i) { { i.operator->() } -> auto&&; }
struct __pointer_type<I> {                                   // exposition only
  using type = decltype(declval<I>().operator->());
};
template <class> struct __iterator_traits { };               // exposition only
template <Iterator I> struct __iterator_traits<I> {
  using difference_type = difference_type_t<I>;
  using value_type = void;
  using reference = void;
  using pointer = void;
  using iterator_category = output_iterator_tag;
};
template <InputIterator I> struct __iterator_traits<I> {  // exposition only
  using difference_type = difference_type_t<I>;
  using value_type = value_type_t<I>;
  using reference = reference_t<I>;
  using pointer = typename __pointer_type<I>::type;
  using iterator_category = iterator_category_t<I>;
};
template <class I>
  using iterator_traits = __iterator_traits<I>;
```

2 [*Note:* `iterator_traits` is an alias template to prevent user code from specializing it. — *end note*]

3 [*Example:* To implement a generic `reverse` function, a C++ program can do the following:

```
template <BidirectionalIterator I>
void reverse(I first, I last) {
  difference_type_t<I> n = distance(first, last);
  --n;
  while(n > 0) {
    value_type_t<I> tmp = *first;
    *first++ = *--last;
    *last = tmp;
    n -= 2;
  }
}
```

— *end example*]

21.3.6.2 Standard iterator traits [std2.iterator.stdtraits]

1 To facilitate interoperability between new code using iterators conforming to this document and older code using iterators that conform to the iterator requirements specified in ISO/IEC 14882, three specializations of `std::iterator_traits` are provided to map the newer iterator categories and associated types to the older ones.

```
namespace std {
  template <experimental::ranges::std2::Iterator Out>
```

```
struct iterator_traits<Out> {
  using difference_type   = ~~experimental::ranges~~::std2::difference_type_t<Out>;
  using value_type        = see below;
  using reference         = see below;
  using pointer           = see below;
  using iterator_category = std::output_iterator_tag;
};
```

2 The nested type `value_type` is computed as follows:

(2.1) — If `Out::value_type` is valid and denotes a type, then `std::iterator_traits<Out>::value_type` is `Out::value_type`.

(2.2) — Otherwise, `std::iterator_traits<Out>::value_type` is `void`.

3 The nested type `reference` is computed as follows:

(3.1) — If `Out::reference` is valid and denotes a type, then `std::iterator_traits<Out>::reference` is `Out::reference`.

(3.2) — Otherwise, `std::iterator_traits<Out>::reference` is `void`.

4 The nested type `pointer` is computed as follows:

(4.1) — If `Out::pointer` is valid and denotes a type, then `std::iterator_traits<Out>::pointer` is `Out::pointer`.

(4.2) — Otherwise, `std::iterator_traits<Out>::pointer` is `void`.

```
  template <~~experimental::ranges~~::std2::InputIterator In>
  struct iterator_traits<In> { };

  template <~~experimental::ranges~~::std2::InputIterator In>
    requires ~~experimental::ranges~~::std2::Sentinel<In, In>
  struct iterator_traits<In> {
    using difference_type   = ~~experimental::ranges~~::std2::difference_type_t<In>;
    using value_type        = ~~experimental::ranges~~::std2::value_type_t<In>;
    using reference         = see below;
    using pointer           = see below;
    using iterator_category = see below;
  };
}
```

5 The nested type `reference` is computed as follows:

(5.1) — If `In::reference` is valid and denotes a type, then `std::iterator_traits<In>::reference` is `In::reference`.

(5.2) — Otherwise, `std::iterator_traits<In>::reference` is ~~`experimental::ranges`~~`::std2::reference_t<In>`.

6 The nested type `pointer` is computed as follows:

(6.1) — If `In::pointer` is valid and denotes a type, then `std::iterator_traits<In>::pointer` is `In::pointer`.

(6.2) — Otherwise, `std::iterator_traits<In>::pointer` is ~~`experimental::ranges`~~`::std2::iterator_traits<In>::pointer`.

7 Let type `C` be ~~`experimental::ranges`~~`::std2::iterator_category_t<In>`. The nested type `std::iterator_traits<In>::iterator_category` is computed as follows:

(7.1) — If `C` is the same as or inherits from `std::input_iterator_tag` or `std::output_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `C`.

(7.2) — Otherwise, if ~~`experimental::ranges`~~`::std2::reference_t<In>` is not a reference type, `std::iterator_traits<In>::iterator_category` is `std::input_iterator_tag`.

(7.3) — Otherwise, if `C` is the same as or inherits from ~~`experimental::ranges`~~`::std2::random_access_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `std::random_access_iterator_tag`.

(7.4) — Otherwise, if `C` is the same as or inherits from ~~`experimental::ranges`~~`::std2::bidirectional_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `std::bidirectional_iterator_tag`.

(7.5) — Otherwise, if `C` is the same as or inherits from ~~`experimental::ranges`~~`::std2::forward_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `std::forward_iterator_tag`.

(7.6) — Otherwise, `std::iterator_traits<In>::iterator_category` is `std::input_iterator_tag`.

8 [*Note:* Some implementations may find it necessary to add additional constraints to these partial specializations to prevent them from being considered for types that conform to the iterator requirements specified in ISO/IEC 14882. — *end note*]

21.3.6.3 Standard iterator tags [std2.iterator.tags]

1 It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which can be used as compile time tags for algorithm selection. [*Note:* The preferred way to dispatch to more specialized algorithm implementations is with concept-based overloading. — *end note*] The category tags are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every input iterator of type `I`, `iterator_category_t<I>` shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  struct output_iterator_tag { };
  struct input_iterator_tag { };
  struct forward_iterator_tag : input_iterator_tag { };
  struct bidirectional_iterator_tag : forward_iterator_tag { };
  struct random_access_iterator_tag : bidirectional_iterator_tag { };
}}~~}}~~
```

2 [*Note:* The `output_iterator_tag` is provided for the sake of backward compatibility. — *end note*]

3 [*Example:* For a program-defined iterator `BinaryTreeIterator`, it could be included into the bidirectional iterator category by specializing the `difference_type`, `value_type`, and `iterator_category` templates:

```
template <class T> struct difference_type<BinaryTreeIterator<T>> {
  using type = ptrdiff_t;
};
```

```
template <class T> struct value_type<BinaryTreeIterator<T>> {
  using type = T;
};
template <class T> struct iterator_category<BinaryTreeIterator<T>> {
  using type = bidirectional_iterator_tag;
};
```

— *end example*]

21.3.6.4 Iterator operations [std2.iterator.operations]

1 Since only types that satisfy `RandomAccessIterator` provide the `+` operator, and types that satisfy `SizedSentinel` provide the `-` operator, the library provides customization point objects (20.1.4.2.1.6) `advance`, `distance`, `next`, and `prev`. These customization point objects use `+` and `-` for random access iterators and ranges that satisfy `SizedSentinel` (and are, therefore, constant time for them); for output, input, forward and bidirectional iterators they use `++` to provide linear time implementations.

2 The name `advance` denotes a customization point object (20.1.4.2.1.6). It has the following function call operators:

```
template <Iterator I>
  constexpr void operator()(I& i, difference_type_t<I> n) const;
```

3 *Requires:* `n` shall be negative only for bidirectional iterators.

4 *Effects:* For random access iterators, equivalent to `i += n`. Otherwise, increments (or decrements for negative `n`) iterator `i` by `n`.

```
template <Iterator I, Sentinel<I> S>
  constexpr void operator()(I& i, S bound) const;
```

5 *Requires:* If `Assignable<I&, S>` is not satisfied, `[i,bound)` shall denote a range.

6 *Effects:*

(6.1) — If `Assignable<I&, S>` is satisfied, equivalent to `i = std::move(bound)`.

(6.2) — Otherwise, if `SizedSentinel<S, I>` is satisfied, equivalent to `advance(i, bound - i)`.

(6.3) — Otherwise, increments `i` until `i == bound`.

```
template <Iterator I, Sentinel<I> S>
  constexpr difference_type_t<I> operator()(I& i, difference_type_t<I> n, S bound) const;
```

7 *Requires:* If `n > 0`, `[i,bound)` shall denote a range. If `n == 0`, `[i,bound)` or `[bound,i)` shall denote a range. If `n < 0`, `[bound,i)` shall denote a range and `(BidirectionalIterator<I> && Same<I, S>)` shall be satisfied.

8 *Effects:*

(8.1) — If `SizedSentinel<S, I>` is satisfied:

(8.1.1) — If |`n`| >= |`bound - i`|, equivalent to `advance(i, bound)`.

(8.1.2) — Otherwise, equivalent to `advance(i, n)`.

(8.2) — Otherwise, increments (or decrements for negative `n`) iterator `i` either `n` times or until `i == bound`, whichever comes first.

9 *Returns:* `n -` M, where M is the distance from the starting position of `i` to the ending position.

10 The name `distance` denotes a customization point object. It has the following function call operators:

```
template <Iterator I, Sentinel<I> S>
  constexpr difference_type_t<I> operator()(I first, S last) const;
```

11 *Requires:* `[first,last)` shall denote a range, or `(Same<S, I> && SizedSentinel<S, I>)` shall be satisfied and `[last,first)` shall denote a range.

12 *Effects:* If `SizedSentinel<S, I>` is satisfied, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

```
template <Range R>
  constexpr difference_type_t<iterator_t<R>> operator()(R&& r) const;
```

Effects: Equivalent to: `return distance(~~ranges~~::std2::begin(r), ~~ranges~~::std2::end(r));` (21.4.4)

Remarks: Instantiations of this member function template may be ill-formed if the declarations in `<~~experimental/ranges~~std2/range>` are not in scope at the point of instantiation (17.7.4.1).

```
template <SizedRange R>
  constexpr difference_type_t<iterator_t<R>> operator()(R&& r) const;
```

13 *Effects:* Equivalent to: `return ~~ranges~~::std2::size(r);` (21.4.5.1)

14 *Remarks:* Instantiations of this member function template may be ill-formed if the declarations in `<~~experimental/ranges~~std2/range>` are not in scope at the point of instantiation (17.7.4.1).

15 The name `next` denotes a customization point object. It has the following function call operators:

```
template <Iterator I>
  constexpr I operator()(I x) const;
```

16 *Effects:* Equivalent to: `++x; return x;`

```
template <Iterator I>
  constexpr I operator()(I x, difference_type_t<I> n) const;
```

17 *Effects:* Equivalent to: `advance(x, n); return x;`

```
template <Iterator I, Sentinel<I> S>
  constexpr I operator()(I x, S bound) const;
```

18 *Effects:* Equivalent to: `advance(x, bound); return x;`

```
template <Iterator I, Sentinel<I> S>
  constexpr I operator()(I x, difference_type_t<I> n, S bound) const;
```

19 *Effects:* Equivalent to: `advance(x, n, bound); return x;`

20 The name `prev` denotes a customization point object. It has the following function call operators:

```
template <BidirectionalIterator I>
  constexpr I operator()(I x) const;
```

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

```
template <BidirectionalIterator I>
  constexpr I operator()(I x, difference_type_t<I> n) const;
```

22 *Effects:* Equivalent to: `advance(x, -n); return x;`

```
template <BidirectionalIterator I>
  constexpr I operator()(I x, difference_type_t<I> n, I bound) const;
```

23 *Effects:* Equivalent to: `advance(x, -n, bound); return x;`

21.3.7 Iterator adaptors [std2.iterators.predef]

21.3.7.1 Reverse iterators [std2.iterators.reverse]

1 Class template `reverse_iterator` is an iterator adaptor that iterates from the end of the sequence defined by its underlying iterator to the beginning of that sequence. The fundamental relation between a reverse iterator and its corresponding underlying iterator `i` is established by the identity: `*make_reverse_iterator(i) == *prev(i)`.

21.3.7.1.1 Class template reverse_iterator [std2.reverse.iterator]

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <BidirectionalIterator I>
  class reverse_iterator {
  public:
    using iterator_type = I;
    using difference_type = difference_type_t<I>;
    using value_type = value_type_t<I>;
    using iterator_category = iterator_category_t<I>;
    using reference = reference_t<I>;
    using pointer = I;

    constexpr reverse_iterator();
    explicit constexpr reverse_iterator(I x);
    template <ConvertibleTo<I> U>
      constexpr reverse_iterator(const reverse_iterator<ConvertibleTo<I>U>& i);
    template <ConvertibleTo<I> U>
      constexpr reverse_iterator& operator=(const reverse_iterator<ConvertibleTo<I>U>& i);

    constexpr I base() const;
    constexpr reference operator*() const;
    constexpr pointer operator->() const;

    constexpr reverse_iterator& operator++();
    constexpr reverse_iterator  operator++(int);
    constexpr reverse_iterator& operator--();
    constexpr reverse_iterator  operator--(int);

    constexpr reverse_iterator  operator+ (difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr reverse_iterator& operator+=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr reverse_iterator  operator- (difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr reverse_iterator& operator-=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr reference operator[](difference_type n) const
      requires RandomAccessIterator<I>;

    friend constexpr rvalue_reference_t<I> iter_move(const reverse_iterator& i)
      noexcept(see below);
    template <IndirectlySwappable<I> I2>
      friend constexpr void iter_swap(const reverse_iterator& x, const reverse_iterator<I2>& y)
        noexcept(see below);

  private:
    I current; // exposition only
```

```
  };

  template <class I1, class I2>
      requires EqualityComparableWith<I1, I2>
    constexpr bool operator==(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <class I1, class I2>
      requires EqualityComparableWith<I1, I2>
    constexpr bool operator!=(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator<(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator>(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator>=(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator<=(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <class I1, class I2>
      requires SizedSentinel<I1, I2>
    constexpr difference_type_t<I2> operator-(
      const reverse_iterator<I1>& x,
      const reverse_iterator<I2>& y);
  template <RandomAccessIterator I>
    constexpr reverse_iterator<I> operator+(
      difference_type_t<I> n,
      const reverse_iterator<I>& x);

  template <BidirectionalIterator I>
    constexpr reverse_iterator<I> make_reverse_iterator(I i);
```

}}~~}}~~

21.3.7.1.2 reverse_iterator operations [std2.reverse.iter.ops]

21.3.7.1.2.1 reverse_iterator constructor [std2.reverse.iter.cons]

`constexpr reverse_iterator();`

1 *Effects:* Value-initializes `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `I`.

`explicit constexpr reverse_iterator(I x);`

2 *Effects:* Initializes `current` with `x`.

`template<ConvertibleTo<I> U>`
`constexpr reverse_iterator(const reverse_iterator<`~~`ConvertibleTo<I>`~~`U>& i);`

3 *Effects:* Initializes `current` with `i.current`.

21.3.7.1.2.2 `reverse_iterator::operator=` [std2.reverse.iter.op=]

`template<ConvertibleTo<I> U>`
`constexpr reverse_iterator&`
`operator=(const reverse_iterator<`~~`ConvertibleTo<I>`~~`U>& i);`

1 *Effects:* Assigns `i.current` to `current`.

2 *Returns:* `*this`.

21.3.7.1.2.3 Conversion [std2.reverse.iter.conv]

```
constexpr I base() const;
```

1 *Returns:* `current`.

21.3.7.1.2.4 `operator*` [std2.reverse.iter.op.star]

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

1 *Effects:* Equivalent to: `return *prev(current);`

21.3.7.1.2.5 `operator->` [std2.reverse.iter.opref]

```
constexpr pointer operator->() const;
```

1 *Effects:* Equivalent to: `return prev(current);`

21.3.7.1.2.6 `operator++` [std2.reverse.iter.op++]

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

1 *Effects:* `--current;`

2 *Returns:* `*this`.

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

3 *Effects:*

```
reverse_iterator tmp = *this;
--current;
return tmp;
```

21.3.7.1.2.7 `operator--` [std2.reverse.iter.op--]

```
constexpr reverse_iterator& operator--();
```

1 *Effects:* `++current`

2 *Returns:* `*this`.

```
constexpr reverse_iterator operator--(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
++current;
return tmp;
```

21.3.7.1.2.8 operator+ [std2.reverse.iter.op+]

```
constexpr reverse_iterator
  operator+(difference_type n) const
    requires RandomAccessIterator<I>;
```

1 *Returns:* `reverse_iterator(current-n)`.

21.3.7.1.2.9 operator+= [std2.reverse.iter.op+=]

```
constexpr reverse_iterator&
  operator+=(difference_type n)
    requires RandomAccessIterator<I>;
```

1 *Effects:* `current -= n;`

2 *Returns:* `*this`.

21.3.7.1.2.10 operator- [std2.reverse.iter.op-]

```
constexpr reverse_iterator
  operator-(difference_type n) const
    requires RandomAccessIterator<I>;
```

1 *Returns:* `reverse_iterator(current+n)`.

21.3.7.1.2.11 operator-= [std2.reverse.iter.op-=]

```
constexpr reverse_iterator&
  operator-=(difference_type n)
    requires RandomAccessIterator<I>;
```

1 *Effects:* `current += n;`

2 *Returns:* `*this`.

21.3.7.1.2.12 operator[] [std2.reverse.iter.opindex]

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

1 *Returns:* `current[-n-1]`.

21.3.7.1.2.13 operator== [std2.reverse.iter.op==]

```
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator==(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current == y.current;`

21.3.7.1.2.14 operator!= [std2.reverse.iter.op!=]

```
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator!=(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current != y.current;`

21.3.7.1.2.15 `operator<` [std2.reverse.iter.op<]

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current > y.current;`

21.3.7.1.2.16 `operator>` [std2.reverse.iter.op>]

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator>(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current < y.current;`

21.3.7.1.2.17 `operator>=` [std2.reverse.iter.op>=]

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator>=(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current <= y.current;`

21.3.7.1.2.18 `operator<=` [std2.reverse.iter.op<=]

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<=(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current >= y.current;`

21.3.7.1.2.19 `operator-` [std2.reverse.iter.opdiff]

```
template <class I1, class I2>
    requires SizedSentinel<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return y.current - x.current;`

21.3.7.1.2.20 `operator+` [std2.reverse.iter.opsum]

```
template <RandomAccessIterator I>
  constexpr reverse_iterator<I> operator+(
    difference_type_t<I> n,
    const reverse_iterator<I>& x);
```

1 *Effects:* Equivalent to: `return reverse_iterator<I>(x.current - n);`

21.3.7.1.2.21 `iter_move` [std2.reverse.iter.iter_move]

```
friend constexpr rvalue_reference_t<I> iter_move(const reverse_iterator& i)
  noexcept(see below);
```

1 *Effects:* Equivalent to: `return` ~~`ranges`~~`::std2::iter_move(prev(i.current));`

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

noexcept(~~ranges~~::std2::iter_move(declval<I&>())) && noexcept(--declval<I&>()) &&
  is_nothrow_copy_constructible_v<I>~~::value~~

21.3.7.1.2.22 iter_swap [std2.reverse.iter.iter_swap]

```
template <IndirectlySwappable<I> I2>
  friend constexpr void iter_swap(const reverse_iterator& x, const reverse_iterator<I2>& y)
    noexcept(see below);
```

1 *Effects:* Equivalent to ~~`ranges`~~`::std2::iter_swap(prev(x.current), prev(y.current))`.

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

noexcept(~~ranges~~::std2::iter_swap(declval<I>(), declval<I>())) && noexcept(--declval<I&>())

21.3.7.1.2.23 Non-member function make_reverse_iterator() [std2.reverse.iter.make]

```
template <BidirectionalIterator I>
  constexpr reverse_iterator<I> make_reverse_iterator(I i);
```

1 *Returns:* `reverse_iterator<I>(i)`.

21.3.7.2 Insert iterators [std2.iterators.insert]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range `[first,last)` to be copied into a range starting with result. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite* mode.

2 An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy `OutputIterator`. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

21.3.7.2.1 Class template back_insert_iterator [std2.back.insert.iterator]

namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {

```
  template <class Container>
  class back_insert_iterator {
  public:
    using container_type = Container;
    using difference_type = ptrdiff_t;

    constexpr back_insert_iterator();
```

```
    explicit back_insert_iterator(Container& x);
    back_insert_iterator&
      operator=(const value_type_t<Container>& value);
    back_insert_iterator&
      operator=(value_type_t<Container>&& value);

    back_insert_iterator& operator*();
    back_insert_iterator& operator++();
    back_insert_iterator operator++(int);

  private:
    Container* container; // exposition only
  };

  template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);
}}}}
```

21.3.7.2.2 `back_insert_iterator` operations [std2.back.insert.iter.ops]

21.3.7.2.2.1 `back_insert_iterator` constructor [std2.back.insert.iter.cons]

```
constexpr back_insert_iterator();
```

1 *Effects:* Value-initializes `container`.

```
explicit back_insert_iterator(Container& x);
```

2 *Effects:* Initializes `container` with `addressof(x)`.

21.3.7.2.2.2 `back_insert_iterator::operator=` [std2.back.insert.iter.op=]

```
back_insert_iterator&
  operator=(const value_type_t<Container>& value);
```

1 *Effects:* Equivalent to `container->push_back(value)`.

2 *Returns:* `*this`.

```
back_insert_iterator&
  operator=(value_type_t<Container>&& value);
```

3 *Effects:* Equivalent to `container->push_back(std::move(value))`.

4 *Returns:* `*this`.

21.3.7.2.2.3 `back_insert_iterator::operator*` [std2.back.insert.iter.op*]

```
back_insert_iterator& operator*();
```

1 *Returns:* `*this`.

21.3.7.2.2.4 `back_insert_iterator::operator++` [std2.back.insert.iter.op++]

```
back_insert_iterator& operator++();
back_insert_iterator operator++(int);
```

1 *Returns:* `*this`.

21.3.7.2.2.5 `back_inserter` [std2.back.inserter]

```
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);
```

1 *Returns:* `back_insert_iterator<Container>(x)`.

21.3.7.2.3 Class template `front_insert_iterator` [std2.front.insert.iterator]

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <class Container>
  class front_insert_iterator {
  public:
    using container_type = Container;
    using difference_type = ptrdiff_t;

    constexpr front_insert_iterator();
    explicit front_insert_iterator(Container& x);
    front_insert_iterator&
      operator=(const value_type_t<Container>& value);
    front_insert_iterator&
      operator=(value_type_t<Container>&& value);

    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);

  private:
    Container* container; // exposition only
  };

  template <class Container>
    front_insert_iterator<Container> front_inserter(Container& x);
}}}}
```

21.3.7.2.4 `front_insert_iterator` operations [std2.front.insert.iter.ops]

21.3.7.2.4.1 `front_insert_iterator` constructor [std2.front.insert.iter.cons]

```
constexpr front_insert_iterator();
```

1 *Effects:* Value-initializes `container`.

```
explicit front_insert_iterator(Container& x);
```

2 *Effects:* Initializes `container` with `addressof(x)`.

21.3.7.2.4.2 `front_insert_iterator::operator=` [std2.front.insert.iter.op=]

```
front_insert_iterator&
  operator=(const value_type_t<Container>& value);
```

1 *Effects:* Equivalent to `container->push_front(value)`.

2 *Returns:* `*this`.

```
front_insert_iterator&
  operator=(value_type_t<Container>&& value);
```

3 *Effects:* Equivalent to `container->push_front(std::move(value))`.

4 *Returns:* `*this`.

21.3.7.2.4.3 `front_insert_iterator::operator*` [std2.front.insert.iter.op*]

```
front_insert_iterator& operator*();
```

1 *Returns:* `*this`.

21.3.7.2.4.4 `front_insert_iterator::operator++` [std2.front.insert.iter.op++]

```
front_insert_iterator& operator++();
front_insert_iterator operator++(int);
```

1 *Returns:* `*this`.

21.3.7.2.4.5 `front_inserter` [std2.front.inserter]

```
template <class Container>
  front_insert_iterator<Container> front_inserter(Container& x);
```

1 *Returns:* `front_insert_iterator<Container>(x)`.

21.3.7.2.5 Class template `insert_iterator` [std2.insert.iterator]

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <class Container>
  class insert_iterator {
  public:
    using container_type = Container;
    using difference_type = ptrdiff_t;

    insert_iterator();
    insert_iterator(Container& x, iterator_t<Container> i);
    insert_iterator&
      operator=(const value_type_t<Container>& value);
    insert_iterator&
      operator=(value_type_t<Container>&& value);

    insert_iterator& operator*();
    insert_iterator& operator++();
    insert_iterator& operator++(int);

  private:
    Container* container;      // exposition only
    iterator_t<Container> iter; // exposition only
  };

  template <class Container>
    insert_iterator<Container> inserter(Container& x, iterator_t<Container> i);
}}}}}
```

21.3.7.2.6 `insert_iterator` operations [std2.insert.iter.ops]

21.3.7.2.6.1 `insert_iterator` constructor [std2.insert.iter.cons]

```
insert_iterator();
```

1 *Effects:* Value-initializes `container` and `iter`.

```
insert_iterator(Container& x, iterator_t<Container> i);
```

2 *Requires:* `i` is an iterator into `x`.

3 *Effects:* Initializes `container` with `addressof(x)` and `iter` with `i`.

21.3.7.2.6.2 `insert_iterator::operator=` [std2.insert.iter.op=]

```
insert_iterator&
  operator=(const value_type_t<Container>& value);
```

1 *Effects:* Equivalent to:

```
iter = container->insert(iter, value);
++iter;
```

2 *Returns:* `*this`.

```
insert_iterator&
  operator=(value_type_t<Container>&& value);
```

3 *Effects:* Equivalent to:

```
iter = container->insert(iter, std::move(value));
++iter;
```

4 *Returns:* `*this`.

21.3.7.2.6.3 `insert_iterator::operator*` [std2.insert.iter.op*]

```
insert_iterator& operator*();
```

1 *Returns:* `*this`.

21.3.7.2.6.4 `insert_iterator::operator++` [std2.insert.iter.op++]

```
insert_iterator& operator++();
insert_iterator& operator++(int);
```

1 *Returns:* `*this`.

21.3.7.2.6.5 `inserter` [std2.inserter]

```
template <class Container>
  insert_iterator<Container> inserter(Container& x, iterator_t<Container> i);
```

1 *Returns:* `insert_iterator<Container>(x, i)`.

21.3.7.3 Move iterators and sentinels [std2.iterators.move]

21.3.7.3.1 Class template `move_iterator` [std2.move.iterator]

1 Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator's indirection operator to an rvalue of the value type. Some generic algorithms can be called with move iterators to replace copying with moving.

2 [*Example:*

```
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end());          // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // moves strings into v2
```

— *end example*]

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <InputIterator I>
  class move_iterator {
  public:
    using iterator_type     = I;
    using difference_type   = difference_type_t<I>;
    using value_type        = value_type_t<I>;
    using iterator_category = input_iterator_tag;
    using reference         = rvalue_reference_t<I>;

    constexpr move_iterator();
    explicit constexpr move_iterator(I i);
    template <ConvertibleTo<I> U>
      constexpr move_iterator(const move_iterator<~~ConvertibleTo<I>~~U>& i);
    template <ConvertibleTo<I> U>
      constexpr move_iterator& operator=(const move_iterator<~~ConvertibleTo<I>~~U>& i);

    constexpr I base() const;
    constexpr reference operator*() const;

    constexpr move_iterator& operator++();
    constexpr void operator++(int);
    constexpr move_iterator operator++(int)
      requires ForwardIterator<I>;
    constexpr move_iterator& operator--()
      requires BidirectionalIterator<I>;
    constexpr move_iterator operator--(int)
      requires BidirectionalIterator<I>;

    constexpr move_iterator operator+(difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr move_iterator& operator+=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr move_iterator operator-(difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr move_iterator& operator-=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr reference operator[](difference_type n) const
      requires RandomAccessIterator<I>;

    friend constexpr rvalue_reference_t<I> iter_move(const move_iterator& i)
      noexcept(see below);
    template <IndirectlySwappable<I> I2>
      friend constexpr void iter_swap(const move_iterator& x, const move_iterator<I2>& y)
        noexcept(see below);

  private:
    I current; // exposition only
  };

  template <class I1, class I2>
      requires EqualityComparableWith<I1, I2>
    constexpr bool operator==(
      const move_iterator<I1>& x, const move_iterator<I2>& y);
  template <class I1, class I2>
```

```
      requires EqualityComparableWith<I1, I2>
    constexpr bool operator!=(
      const move_iterator<I1>& x, const move_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator<(
      const move_iterator<I1>& x, const move_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator<=(
      const move_iterator<I1>& x, const move_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator>(
      const move_iterator<I1>& x, const move_iterator<I2>& y);
  template <class I1, class I2>
      requires StrictTotallyOrderedWith<I1, I2>
    constexpr bool operator>=(
      const move_iterator<I1>& x, const move_iterator<I2>& y);

  template <class I1, class I2>
      requires SizedSentinel<I1, I2>
    constexpr difference_type_t<I2> operator-(
      const move_iterator<I1>& x,
      const move_iterator<I2>& y);
  template <RandomAccessIterator I>
    constexpr move_iterator<I> operator+(
      difference_type_t<I> n,
      const move_iterator<I>& x);
  template <InputIterator I>
    constexpr move_iterator<I> make_move_iterator(I i);
}}~~}}~~
```

3 [*Note:* `move_iterator` does not provide an `operator->` because the class member access expression `i->m` may have different semantics than the expression `(*i).m` when the expression `*i` is an rvalue. — *end note*]

21.3.7.3.2 `move_iterator` operations [std2.move.iter.ops]

21.3.7.3.2.1 `move_iterator` constructors [std2.move.iter.op.const]

```
constexpr move_iterator();
```

1 *Effects:* Constructs a `move_iterator`, value-initializing `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `I`.

```
explicit constexpr move_iterator(I i);
```

2 *Effects:* Constructs a `move_iterator`, initializing `current` with `i`.

```
template <ConvertibleTo<I> U>
  constexpr move_iterator(const move_iterator<~~ConvertibleTo<I>~~U>& i);
```

3 *Effects:* Constructs a `move_iterator`, initializing `current` with `i.current`.

21.3.7.3.2.2 `move_iterator::operator=` [std2.move.iter.op=]

`template <ConvertibleTo<I> U>`
`  constexpr move_iterator& operator=(const move_iterator<`~~`ConvertibleTo<I>`~~`U>& i);`

1 *Effects:* Assigns `i.current` to `current`.

21.3.7.3.2.3 `move_iterator` conversion [std2.move.iter.op.conv]

```
constexpr I base() const;
```

1 *Returns:* `current`.

21.3.7.3.2.4 `move_iterator::operator*` [std2.move.iter.op.star]

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

1 *Effects:* Equivalent to: `return iter_move(current);`

21.3.7.3.2.5 `move_iterator::operator++` [std2.move.iter.op.incr]

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

1 *Effects:* Equivalent to `++current`.

2 *Returns:* `*this`.

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

3 *Effects:* Equivalent to `++current`.

```
constexpr move_iterator operator++(int)
  requires ForwardIterator<I>;
```

4 *Effects:* Equivalent to:

```
move_iterator tmp = *this;
++current;
return tmp;
```

21.3.7.3.2.6 `move_iterator::operator--` [std2.move.iter.op.decr]

```
constexpr move_iterator& operator--()
  requires BidirectionalIterator<I>;
```

1 *Effects:* Equivalent to `--current`.

2 *Returns:* `*this`.

```
constexpr move_iterator operator--(int)
  requires BidirectionalIterator<I>;
```

3 *Effects:* Equivalent to:

```
move_iterator tmp = *this;
--current;
return tmp;
```

21.3.7.3.2.7 `move_iterator::operator+` [std2.move.iter.op.+]

```
constexpr move_iterator operator+(difference_type n) const
  requires RandomAccessIterator<I>;
```

1 *Effects:* Equivalent to: `return move_iterator(current + n);`

21.3.7.3.2.8 `move_iterator::operator+=` [std2.move.iter.op.+=]

```
constexpr move_iterator& operator+=(difference_type n)
  requires RandomAccessIterator<I>;
```

1 *Effects:* Equivalent to `current += n`.

2 *Returns:* `*this`.

21.3.7.3.2.9 `move_iterator::operator-` [std2.move.iter.op.-]

```
constexpr move_iterator operator-(difference_type n) const
  requires RandomAccessIterator<I>;
```

1 *Effects:* Equivalent to: `return move_iterator(current - n);`

21.3.7.3.2.10 `move_iterator::operator-=` [std2.move.iter.op.-=]

```
constexpr move_iterator& operator-=(difference_type n)
  requires RandomAccessIterator<I>;
```

1 *Effects:* Equivalent to `current -= n`.

2 *Returns:* `*this`.

21.3.7.3.2.11 `move_iterator::operator[]` [std2.move.iter.op.index]

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

1 *Effects:* Equivalent to: `return iter_move(current + n);`

21.3.7.3.2.12 `move_iterator` comparisons [std2.move.iter.op.comp]

```
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator==(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current == y.current;`

```
template <class I1, class I2>
    requires EqualityComparableWith<I1, I2>
  constexpr bool operator!=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
```

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

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
```

3 *Effects:* Equivalent to: `return x.current < y.current;`

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator<=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
```

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

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator>(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
```

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

```
template <class I1, class I2>
    requires StrictTotallyOrderedWith<I1, I2>
  constexpr bool operator>=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);
```

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

21.3.7.3.2.13 `move_iterator` non-member functions [std2.move.iter.nonmember]

```
template <class I1, class I2>
    requires SizedSentinel<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const move_iterator<I1>& x,
    const move_iterator<I2>& y);
```

1 *Effects:* Equivalent to: `return x.current - y.current;`

```
template <RandomAccessIterator I>
  constexpr move_iterator<I> operator+(
    difference_type_t<I> n,
    const move_iterator<I>& x);
```

2 *Effects:* Equivalent to: `return x + n;`

```
friend constexpr rvalue_reference_t<I> iter_move(const move_iterator& i)
  noexcept(see below);
```

3 *Effects:* Equivalent to: return ~~ranges~~::std2::iter_move(i.current);

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

noexcept(~~ranges~~::std2::iter_move(i.current))

```
template <IndirectlySwappable<I> I2>
  friend constexpr void iter_swap(const move_iterator& x, const move_iterator<I2>& y)
    noexcept(see below);
```

5 *Effects:* Equivalent to: ~~ranges~~::std2::iter_swap(x.current, y.current).

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

noexcept(~~ranges~~::std2::iter_swap(x.current, y.current))

```
template <InputIterator I>
  constexpr move_iterator<I> make_move_iterator(I i);
```

7 *Returns:* `move_iterator<I>(i)`.

21.3.7.3.3 Class template `move_sentinel` [std2.move.sentinel]

1 Class template `move_sentinel` is a sentinel adaptor useful for denoting ranges together with `move_iterator`. When an input iterator type `I` and sentinel type `S` satisfy `Sentinel<S, I>`, `Sentinel<move_sentinel<S>, move_iterator<I>>` is satisfied as well.

2 [*Example:* A `move_if` algorithm is easily implemented with `copy_if` using `move_iterator` and `move_sentinel`:

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
          IndirectUnaryPredicate<I> Pred>
  requires IndirectlyMovable<I, O>
void move_if(I first, S last, O out, Pred pred)
{
  copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}
```

— *end example*]

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <Semiregular S>
  class move_sentinel {
  public:
    constexpr move_sentinel();
    explicit move_sentinel(S s);
    template <ConvertibleTo<S> U>
      move_sentinel(const move_sentinel<~~ConvertibleTo<S>~~U>& s);
    template <ConvertibleTo<S> U>
      move_sentinel& operator=(const move_sentinel<~~ConvertibleTo<S>~~U>& s);

    S base() const;

  private:
    S last; // exposition only
  };

  template <class I, Sentinel<I> S>
    constexpr bool operator==(
      const move_iterator<I>& i, const move_sentinel<S>& s);
  template <class I, Sentinel<I> S>
    constexpr bool operator==(
      const move_sentinel<S>& s, const move_iterator<I>& i);
  template <class I, Sentinel<I> S>
    constexpr bool operator!=(
      const move_iterator<I>& i, const move_sentinel<S>& s);
  template <class I, Sentinel<I> S>
    constexpr bool operator!=(
      const move_sentinel<S>& s, const move_iterator<I>& i);

  template <class I, SizedSentinel<I> S>
    constexpr difference_type_t<I> operator-(
      const move_sentinel<S>& s, const move_iterator<I>& i);
  template <class I, SizedSentinel<I> S>
    constexpr difference_type_t<I> operator-(
      const move_iterator<I>& i, const move_sentinel<S>& s);

  template <Semiregular S>
```

```
    constexpr move_sentinel<S> make_move_sentinel(S s);
}}}}
```

21.3.7.3.4 move_sentinel operations [std2.move.sent.ops]

21.3.7.3.4.1 move_sentinel constructors [std2.move.sent.op.const]

```
constexpr move_sentinel();
```

1 *Effects:* Constructs a `move_sentinel`, value-initializing `last`. If `is_trivially_default_constructible_v<S>~~::value~~` is `true`, then this constructor is a `constexpr` constructor.

```
explicit move_sentinel(S s);
```

2 *Effects:* Constructs a `move_sentinel`, initializing `last` with `s`.

```
template <ConvertibleTo<S> U>
  move_sentinel(const move_sentinel<~~ConvertibleTo<S>~~U>& s);
```

3 *Effects:* Constructs a `move_sentinel`, initializing `last` with `s.last`.

21.3.7.3.4.2 move_sentinel::operator= [std2.move.sent.op=]

```
template <ConvertibleTo<S> U>
  move_sentinel& operator=(const move_sentinel<~~ConvertibleTo<S>~~U>& s);
```

1 *Effects:* Assigns `s.last` to `last`.

2 *Returns:* `*this`.

21.3.7.3.4.3 move_sentinel comparisons [std2.move.sent.op.comp]

```
template <class I, Sentinel<I> S>
  constexpr bool operator==(
    const move_iterator<I>& i, const move_sentinel<S>& s);
template <class I, Sentinel<I> S>
  constexpr bool operator==(
    const move_sentinel<S>& s, const move_iterator<I>& i);
```

1 *Effects:* Equivalent to: `return i.current == s.last;`

```
template <class I, Sentinel<I> S>
  constexpr bool operator!=(
    const move_iterator<I>& i, const move_sentinel<S>& s);
template <class I, Sentinel<I> S>
  constexpr bool operator!=(
    const move_sentinel<S>& s, const move_iterator<I>& i);
```

2 *Effects:* Equivalent to: `return !(i == s);`

21.3.7.3.4.4 move_sentinel non-member functions [std2.move.sent.nonmember]

```
template <class I, SizedSentinel<I> S>
  constexpr difference_type_t<I> operator-(
    const move_sentinel<S>& s, const move_iterator<I>& i);
```

1 *Effects:* Equivalent to: `return s.last - i.current;`

```
template <class I, SizedSentinel<I> S>
  constexpr difference_type_t<I> operator-(
    const move_iterator<I>& i, const move_sentinel<S>& s);
```

2 *Effects:* Equivalent to: `return i.current - s.last;`

```
template <Semiregular S>
  constexpr move_sentinel<S> make_move_sentinel(S s);
```

3 *Returns:* `move_sentinel<S>(s)`.

21.3.7.4 Common iterators [std2.iterators.common]

[Editor's note: TODO: respecify this in terms of `std::variant`.]

1 Class template `common_iterator` is an iterator/sentinel adaptor that is capable of representing a non-bounded range of elements (where the types of the iterator and sentinel differ) as a bounded range (where they are the same). It does this by holding either an iterator or a sentinel, and implementing the equality comparison operators appropriately.

2 [*Note:* The `common_iterator` type is useful for interfacing with legacy code that expects the begin and end of a range to have the same type. — *end note*]

3 [*Example:*

```
template <class ForwardIterator>
void fun(ForwardIterator begin, ForwardIterator end);

list<int> s;
// populate the list s
using CI =
  common_iterator<counted_iterator<list<int>::iterator>,
                  default_sentinel>;
// call fun on a range of 10 ints
fun(CI(make_counted_iterator(s.begin(), 10)),
    CI(default_sentinel()));
```

— *end example*]

21.3.7.4.1 Class template `common_iterator` [std2.common.iterator]

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <Iterator I, Sentinel<I> S>
    requires !Same<I, S>
  class common_iterator {
  public:
    using difference_type = difference_type_t<I>;

    constexpr common_iterator();
    constexpr common_iterator(I i);
    constexpr common_iterator(S s);
    constexpr common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
    template <ConvertibleTo<I> II, ConvertibleTo<S> SS>
      constexpr common_iterator(const common_iterator<II, SS>& u);
    common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
    template <ConvertibleTo<I> II, ConvertibleTo<S> SS>
      common_iterator& operator=(const common_iterator<II, SS>& u);

    decltype(auto) operator*();
    decltype(auto) operator*() const
      requires dereferenceable <const I>;
    decltype(auto) operator->() const
      requires see below;
```

```
    common_iterator& operator++();
    decltype(auto) operator++(int);
    common_iterator operator++(int)
      requires ForwardIterator<I>;

    friend rvalue_reference_t<I> iter_move(const common_iterator& i)
      noexcept(see below)
        requires InputIterator<I>;
    template <IndirectlySwappable<I> I2, class S2>
      friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
        noexcept(see below);

  private:
    bool is_sentinel; // exposition only
    I iter;           // exposition only
    S sentinel;       // exposition only
  };

  template <Readable I, class S>
  struct value_type<common_iterator<I, S>> {
    using type = value_type_t<I>;
  };

  template <InputIterator I, class S>
  struct iterator_category<common_iterator<I, S>> {
    using type = input_iterator_tag;
  };

  template <ForwardIterator I, class S>
  struct iterator_category<common_iterator<I, S>> {
    using type = forward_iterator_tag;
  };

  template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  bool operator==(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
  template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
    requires EqualityComparableWith<I1, I2>
  bool operator==(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
  template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  bool operator!=(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

  template <class I2, SizedSentinel<I2> I1, SizedSentinel<I2> S1, SizedSentinel<I1> S2>
  difference_type_t<I2> operator-(
    const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
}}}}
```

21.3.7.4.2 common_iterator operations [std2.common.iter.ops]

21.3.7.4.2.1 common_iterator constructors [std2.common.iter.op.const]

```
constexpr common_iterator();
```

1 *Effects:* Constructs a `common_iterator`, value-initializing `is_sentinel`, `iter`, and `sentinel`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `I`.

```
constexpr common_iterator(I i);
```

2 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `false`, `iter` with `i`, and value-initializing `sentinel`.

```
constexpr common_iterator(S s);
```

3 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `true`, value-initializing `iter`, and initializing `sentinel` with `s`.

```
constexpr common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
template <ConvertibleTo<I> II, ConvertibleTo<S> SS>
  constexpr common_iterator(const common_iterator<II, SS>& u);
```

4 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `u.is_sentinel`, `iter` with `u.iter`, and `sentinel` with `u.sentinel`.

21.3.7.4.2.2 `common_iterator::operator=` [std2.common.iter.op=]

```
common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
template <ConvertibleTo<I> II, ConvertibleTo<S> SS>
  common_iterator& operator=(const common_iterator<II, SS>& u);
```

1 *Effects:* Assigns `u.is_sentinel` to `is_sentinel`, `u.iter` to `iter`, and `u.sentinel` to `sentinel`.

2 *Returns:* `*this`

21.3.7.4.2.3 `common_iterator::operator*` [std2.common.iter.op.star]

```
decltype(auto) operator*();
decltype(auto) operator*() const
  requires dereferenceable <const I>;
```

1 *Requires:* `!is_sentinel`

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

21.3.7.4.2.4 `common_iterator::operator->` [std2.common.iter.op.ref]

```
decltype(auto) operator->() const
  requires see below;
```

1 *Requires:* `!is_sentinel`

2 *Effects:* Equivalent to:

(2.1) — If `I` is a pointer type or if the expression `i.operator->()` is well-formed, `return iter;`

(2.2) — Otherwise, if the expression `*iter` is a glvalue:

```
auto&& tmp = *iter;
return addressof(tmp);
```

(2.3) — Otherwise, `return proxy(*iter);` where `proxy` is the exposition-only class:

```
class proxy {                // exposition only
  value_type_t<I> keep_;
  proxy(reference_t<I>&& x)
    : keep_(std::move(x)) {}
```

```
public:
  const value_type_t<I>* operator->() const {
    return addressof(keep_);
  }
};
```

3 The expression in the requires clause is equivalent to:

```
Readable<const I> &&
  (requires(const I& i) { i.operator->(); } ||
   is_reference_v<reference_t<I>>::value ||
   Constructible<value_type_t<I>, reference_t<I>>)
```

21.3.7.4.2.5 `common_iterator::operator++` [std2.common.iter.op.incr]

```
common_iterator& operator++();
```

1 *Requires:* `!is_sentinel`

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

3 *Returns:* `*this`.

```
decltype(auto) operator++(int);
```

4 *Requires:* `!is_sentinel`.

5 *Effects:* Equivalent to: `return iter++;`

```
common_iterator operator++(int)
  requires ForwardIterator<I>;
```

6 *Requires:* `!is_sentinel`

7 *Effects:* Equivalent to:

```
common_iterator tmp = *this;
++iter;
return tmp;
```

21.3.7.4.2.6 `common_iterator` comparisons [std2.common.iter.op.comp]

```
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator==(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

1 *Effects:* Equivalent to:

```
return x.is_sentinel ?
  (y.is_sentinel || y.iter == x.sentinel) :
  (!y.is_sentinel || x.iter == y.sentinel);
```

```
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
  requires EqualityComparableWith<I1, I2>
bool operator==(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

2 *Effects:* Equivalent to:

```
return x.is_sentinel ?
  (y.is_sentinel || y.iter == x.sentinel) :
  (y.is_sentinel ?
      x.iter == y.sentinel :
      x.iter == y.iter);
```

```
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator!=(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

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

```
template <class I2, SizedSentinel<I2> I1, SizedSentinel<I2> S1, SizedSentinel<I1> S2>
difference_type_t<I2> operator-(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

4 *Effects:* Equivalent to:

```
return x.is_sentinel ?
  (y.is_sentinel ? 0 : x.sentinel - y.iter) :
  (y.is_sentinel ?
       x.iter - y.sentinel :
       x.iter - y.iter);
```

21.3.7.4.2.7 iter_move [std2.common.iter.op.iter_move]

```
friend rvalue_reference_t<I> iter_move(const common_iterator& i)
  noexcept(see below)
    requires InputIterator<I>;
```

1 *Requires:* `!i.is_sentinel`.

2 *Effects:* Equivalent to: `return` ~~ranges~~`::std2::iter_move(i.iter);`

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

`noexcept(`~~ranges~~`::std2::iter_move(i.iter))`

21.3.7.4.2.8 iter_swap [std2.common.iter.op.iter_swap]

```
template <IndirectlySwappable<I> I2>
  friend void iter_swap(const common_iterator& x, const common_iterator<I2>& y)
    noexcept(see below);
```

1 *Requires:* `!x.is_sentinel && !y.is_sentinel`.

2 *Effects:* Equivalent to ~~ranges~~`::std2::iter_swap(x.iter, y.iter)`.

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

`noexcept(`~~ranges~~`::std2::iter_swap(x.iter, y.iter))`

21.3.7.5 Default sentinels [std2.default.sentinels]

21.3.7.5.1 Class default_sentinel [std2.default.sent]

`namespace` ~~std { namespace experimental { namespace ranges~~`std2 { inline namespace v1 {`
`  class default_sentinel { };`
`}}`~~}}~~

1 Class `default_sentinel` is an empty type used to denote the end of a range. It is intended to be used together with iterator types that know the bound of their range (e.g., `counted_iterator` (21.3.7.6.1)).

21.3.7.6 Counted iterators [std2.iterators.counted]

21.3.7.6.1 Class template `counted_iterator` [std2.counted.iterator]

1 Class template `counted_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of its distance from its starting position. It can be used together with class `default_sentinel` in calls to generic algorithms to operate on a range of N elements starting at a given position without needing to know the end position *a priori*.

2 [*Example:*

```
list<string> s;
// populate the list s with at least 10 strings
vector<string> v(make_counted_iterator(s.begin(), 10),
                 default_sentinel()); // copies 10 strings into v
```

— *end example*]

3 Two values `i1` and `i2` of (possibly differing) types `counted_iterator<I1>` and `counted_iterator<I2>` refer to elements of the same sequence if and only if `next(i1.base(), i1.count())` and `next(i2.base(), i2.count())` refer to the same (possibly past-the-end) element.

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <Iterator I>
  class counted_iterator {
  public:
    using iterator_type = I;
    using difference_type = difference_type_t<I>;

    constexpr counted_iterator();
    constexpr counted_iterator(I x, difference_type_t<I> n);
    template <ConvertibleTo<I> U>
      constexpr counted_iterator(const counted_iterator<ConvertibleTo<I>U>& i);
    template <ConvertibleTo<I> U>
      constexpr counted_iterator& operator=(const counted_iterator<ConvertibleTo<I>U>& i);

    constexpr I base() const;
    constexpr difference_type_t<I> count() const;
    constexpr decltype(auto) operator*();
    constexpr decltype(auto) operator*() const
      requires dereferenceable <const I>;

    constexpr counted_iterator& operator++();
    decltype(auto) operator++(int);
    constexpr counted_iterator operator++(int)
      requires ForwardIterator<I>;
    constexpr counted_iterator& operator--()
      requires BidirectionalIterator<I>;
    constexpr counted_iterator operator--(int)
      requires BidirectionalIterator<I>;

    constexpr counted_iterator  operator+ (difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr counted_iterator& operator+=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr counted_iterator  operator- (difference_type n) const
```

```
    requires RandomAccessIterator<I>;
  constexpr counted_iterator& operator-=(difference_type n)
    requires RandomAccessIterator<I>;
  constexpr decltype(auto) operator[](difference_type n) const
    requires RandomAccessIterator<I>;

  friend constexpr rvalue_reference_t<I> iter_move(const counted_iterator& i)
    noexcept(see below)
      requires InputIterator<I>;
  template <IndirectlySwappable<I> I2>
    friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
      noexcept(see below);

private:
  I current; // exposition only
  difference_type_t<I> cnt; // exposition only
};

template <Readable I>
struct value_type<counted_iterator<I>> {
  using type = value_type_t<I>;
};

template <InputIterator I>
struct iterator_category<counted_iterator<I>> {
  using type = iterator_category_t<I>;
};

template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I>
  constexpr bool operator==(
    const counted_iterator<~~auto~~I>& x, default_sentinel);
template <class I>
  constexpr bool operator==(
    default_sentinel, const counted_iterator<~~auto~~I>& x);

template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I>
    constexpr bool operator!=(
    const counted_iterator<~~auto~~I>& x, default_sentinel y);
template <class I>
  constexpr bool operator!=(
    default_sentinel x, const counted_iterator<~~auto~~I>& y);

template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I1, class I2>
```

```
      requires Common<I1, I2>
    constexpr bool operator<=(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <class I1, class I2>
      requires Common<I1, I2>
    constexpr bool operator>(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <class I1, class I2>
      requires Common<I1, I2>
    constexpr bool operator>=(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <class I1, class I2>
      requires Common<I1, I2>
    constexpr difference_type_t<I2> operator-(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <class I>
    constexpr difference_type_t<I> operator-(
      const counted_iterator<I>& x, default_sentinel y);
  template <class I>
    constexpr difference_type_t<I> operator-(
      default_sentinel x, const counted_iterator<I>& y);

  template <RandomAccessIterator I>
    constexpr counted_iterator<I> operator+(
      difference_type_t<I> n, const counted_iterator<I>& x);

  template <Iterator I>
    constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);
}}}}
```

21.3.7.6.2 `counted_iterator` operations [std2.counted.iter.ops]

21.3.7.6.2.1 `counted_iterator` constructors [std2.counted.iter.op.const]

`constexpr counted_iterator();`

1 *Effects:* Constructs a `counted_iterator`, value-initializing `current` and `cnt`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type `I`.

`constexpr counted_iterator(I i, difference_type_t<I> n);`

2 *Requires:* `n >= 0`

3 *Effects:* Constructs a `counted_iterator`, initializing `current` with `i` and `cnt` with `n`.

`template <ConvertibleTo<I> U>`
`  constexpr counted_iterator(const counted_iterator<`~~`ConvertibleTo<I>`~~`U>& i);`

4 *Effects:* Constructs a `counted_iterator`, initializing `current` with `i.current` and `cnt` with `i.cnt`.

21.3.7.6.2.2 `counted_iterator::operator=` [std2.counted.iter.op=]

`template <ConvertibleTo<I> U>`
`  constexpr counted_iterator& operator=(const counted_iterator<`~~`ConvertibleTo<I>`~~`U>& i);`

1 *Effects:* Assigns `i.current` to `current` and `i.cnt` to `cnt`.

21.3.7.6.2.3 `counted_iterator` conversion [std2.counted.iter.op.conv]

```
constexpr I base() const;
```

1 *Returns:* `current`.

21.3.7.6.2.4 `counted_iterator` count [std2.counted.iter.op.cnt]

```
constexpr difference_type_t<I> count() const;
```

1 *Returns:* `cnt`.

21.3.7.6.2.5 `counted_iterator::operator*` [std2.counted.iter.op.star]

```
constexpr decltype(auto) operator*();
constexpr decltype(auto) operator*() const
  requires dereferenceable <const I>;
```

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

21.3.7.6.2.6 `counted_iterator::operator++` [std2.counted.iter.op.incr]

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

1 *Requires:* `cnt > 0`

2 *Effects:* Equivalent to:

```
++current;
--cnt;
```

3 *Returns:* `*this`.

```
decltype(auto) operator++(int);
```

4 *Requires:* `cnt > 0`.

5 *Effects:* Equivalent to:

```
--cnt;
try { return current++; }
catch(...) { ++cnt; throw; }
```

```
constexpr counted_iterator operator++(int)
  requires ForwardIterator<I>;
```

6 *Requires:* `cnt > 0`

7 *Effects:* Equivalent to:

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

21.3.7.6.2.7 `counted_iterator::operator--` [std2.counted.iter.op.decr]

```
constexpr counted_iterator& operator--();
  requires BidirectionalIterator<I>
```

1 *Effects:* Equivalent to:

```
--current;
++cnt;
```

2 *Returns:* `*this`.

```
constexpr counted_iterator operator--(int)
  requires BidirectionalIterator<I>;
```

3 *Effects:* Equivalent to:

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

21.3.7.6.2.8 `counted_iterator::operator+` [std2.counted.iter.op.+]

```
constexpr counted_iterator operator+(difference_type n) const
  requires RandomAccessIterator<I>;
```

1 *Requires:* `n <= cnt`

2 *Effects:* Equivalent to: `return counted_iterator(current + n, cnt - n);`

21.3.7.6.2.9 `counted_iterator::operator+=` [std2.counted.iter.op.+=]

```
constexpr counted_iterator& operator+=(difference_type n)
  requires RandomAccessIterator<I>;
```

1 *Requires:* `n <= cnt`

2 *Effects:*

```
current += n;
cnt -= n;
```

3 *Returns:* `*this`.

21.3.7.6.2.10 `counted_iterator::operator-` [std2.counted.iter.op.-]

```
constexpr counted_iterator operator-(difference_type n) const
  requires RandomAccessIterator<I>;
```

1 *Requires:* `-n <= cnt`

2 *Effects:* Equivalent to: `return counted_iterator(current - n, cnt + n);`

21.3.7.6.2.11 `counted_iterator::operator-=` [std2.counted.iter.op.-=]

```
constexpr counted_iterator& operator-=(difference_type n)
  requires RandomAccessIterator<I>;
```

1 *Requires:* `-n <= cnt`

2 *Effects:*

```
current -= n;
cnt += n;
```

3 *Returns:* `*this`.

21.3.7.6.2.12 `counted_iterator::operator[]` [std2.counted.iter.op.index]

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

1 *Requires:* `n <= cnt`

2 *Effects:* Equivalent to: `return current[n];`

21.3.7.6.2.13 `counted_iterator` comparisons [std2.counted.iter.op.comp]

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Requires:* `x` and `y` shall refer to elements of the same sequence (21.3.7.6).

2 *Effects:* Equivalent to: `return x.cnt == y.cnt;`

```
template <class I>
  constexpr bool operator==(
    const counted_iterator<~~auto~~I>& x, default_sentinel);
template <class I>
  constexpr bool operator==(
    default_sentinel, const counted_iterator<~~auto~~I>& x);
```

3 *Effects:* Equivalent to: `return x.cnt == 0;`

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <class I>
  constexpr bool operator!=(
    const counted_iterator<~~auto~~I>& x, default_sentinel);
template <class I>
  constexpr bool operator!=(
    default_sentinel, const counted_iterator<~~auto~~I>& x);
```

4 *Requires:* For the first overload, `x` and `y` shall refer to elements of the same sequence (21.3.7.6).

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

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

6 *Requires:* `x` and `y` shall refer to elements of the same sequence (21.3.7.6).

7 *Effects:* Equivalent to: `return y.cnt < x.cnt;`

8 [*Note:* The argument order in the *Effects* element is reversed because `cnt` counts down, not up. — *end note*]

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

9 *Requires:* `x` and `y` shall refer to elements of the same sequence (21.3.7.6).

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

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

11 *Requires:* x and y shall refer to elements of the same sequence (21.3.7.6).

12 *Effects:* Equivalent to: return y < x;

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

13 *Requires:* x and y shall refer to elements of the same sequence (21.3.7.6).

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

21.3.7.6.2.14 counted_iterator non-member functions [std2.counted.iter.nonmember]

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Requires:* x and y shall refer to elements of the same sequence (21.3.7.6).

2 *Effects:* Equivalent to: return y.cnt - x.cnt;

```
template <class I>
  constexpr difference_type_t<I> operator-(
    const counted_iterator<I>& x, default_sentinel y);
```

3 *Effects:* Equivalent to: return -x.cnt;

```
template <class I>
  constexpr difference_type_t<I> operator-(
    default_sentinel x, const counted_iterator<I>& y);
```

4 *Effects:* Equivalent to: return y.cnt;

```
template <RandomAccessIterator I>
  constexpr counted_iterator<I> operator+(
    difference_type_t<I> n, const counted_iterator<I>& x);
```

5 *Requires:* n <= x.cnt.

6 *Effects:* Equivalent to: return x + n;

```
friend constexpr rvalue_reference_t<I> iter_move(const counted_iterator& i)
  noexcept(see below)
    requires InputIterator<I>;
```

7 *Effects:* Equivalent to: return ~~ranges~~::std2::iter_move(i.current);

8 *Remarks:* The expression in noexcept is equivalent to:

noexcept(~~ranges~~::std2::iter_move(i.current))

```
template <IndirectlySwappable<I> I2>
  friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
    noexcept(see below);
```

9 *Effects:* Equivalent to ~~ranges~~::std2::iter_swap(x.current, y.current).

10 *Remarks:* The expression in noexcept is equivalent to:

noexcept(~~ranges~~::std2::iter_swap(x.current, y.current))

```
template <Iterator I>
  constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);
```

11 *Requires:* `n >= 0`.

12 *Returns:* `counted_iterator<I>(i, n)`.

[Editor's note: `dangling` has been moved to the "Ranges library" subclause.]

21.3.7.7 Unreachable sentinel [std2.unreachable.sentinels]

21.3.7.7.1 Class `unreachable` [std2.unreachable.sentinel]

1 Class `unreachable` is a sentinel type that can be used with any `Iterator` to denote an infinite range. Comparing an iterator for equality with an object of type `unreachable` always returns `false`.

[*Example:*

```
char* p;
// set p to point to a character buffer containing newlines
char* nl = find(p, unreachable(), '\n');
```

Provided a newline character really exists in the buffer, the use of `unreachable` above potentially makes the call to `find` more efficient since the loop test against the sentinel does not require a conditional branch. — *end example*]

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  class unreachable { };

  template <Iterator I>
    constexpr bool operator==(const I&, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator==(unreachable, const I&) noexcept;
  template <Iterator I>
    constexpr bool operator!=(const I&, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator!=(unreachable, const I&) noexcept;
}}~~}}~~
```

21.3.7.7.2 `unreachable` operations [std2.unreachable.sentinel.ops]

21.3.7.7.2.1 `operator==` [std2.unreachable.sentinel.op==]

```
template <Iterator I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator==(unreachable, const I&) noexcept;
```

1 *Returns:* `false`.

21.3.7.7.2.2 `operator!=` [std2.unreachable.sentinel.op!=]

```
template <Iterator I>
  constexpr bool operator!=(const I& x, unreachable y) noexcept;
template <Iterator I>
  constexpr bool operator!=(unreachable x, const I& y) noexcept;
```

1 *Returns:* `true`.

21.3.8 Stream iterators [std2.iterators.stream]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[*Example:*

```
partial_sum(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. — *end example*]

21.3.8.1 Class template `istream_iterator` [std2.istream.iterator]

1 The class template `istream_iterator` is an input iterator (21.3.3.11) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

2 Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <class T, class charT = char, class traits = char_traits<charT>,
      class Distance = ptrdiff_t>
  class istream_iterator {
  public:
    typedef input_iterator_tag iterator_category;
    typedef Distance difference_type;
    typedef T value_type;
    typedef const T& reference;
    typedef const T* pointer;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT, traits> istream_type;
    constexpr istream_iterator();
    constexpr istream_iterator(default_sentinel);
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator& x) = default;
    ~istream_iterator() = default;

    const T& operator*() const;
    const T* operator->() const;
    istream_iterator& operator++();
    istream_iterator  operator++(int);
  private:
    basic_istream<charT, traits>* in_stream; // exposition only
    T value;                                 // exposition only
  };
```

```
    template <class T, class charT, class traits, class Distance>
      bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
              const istream_iterator<T, charT, traits, Distance>& y);
    template <class T, class charT, class traits, class Distance>
      bool operator==(default_sentinel x,
              const istream_iterator<T, charT, traits, Distance>& y);
    template <class T, class charT, class traits, class Distance>
      bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
              default_sentinel y);
    template <class T, class charT, class traits, class Distance>
      bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
              const istream_iterator<T, charT, traits, Distance>& y);
    template <class T, class charT, class traits, class Distance>
      bool operator!=(default_sentinel x,
              const istream_iterator<T, charT, traits, Distance>& y);
    template <class T, class charT, class traits, class Distance>
      bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
              default_sentinel y);
  }}}}
```

21.3.8.1.1 istream_iterator constructors and destructor [std2.istream.iterator.cons]

```
constexpr istream_iterator();
constexpr istream_iterator(default_sentinel);
```

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then these constructors shall be `constexpr` constructors.

2 *Postcondition:* `in_stream == nullptr`.

```
istream_iterator(istream_type& s);
```

3 *Effects:* Initializes `in_stream` with `&s`. `value` may be initialized during construction or the first time it is referenced.

4 *Postcondition:* `in_stream == &s`.

```
istream_iterator(const istream_iterator& x) = default;
```

5 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6 *Postcondition:* `in_stream == x.in_stream`.

```
~istream_iterator() = default;
```

7 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

21.3.8.1.2 istream_iterator operations [std2.istream.iterator.ops]

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

1 *Returns:* `value`.

```
const T* operator->() const;
```

2 *Effects:* Equivalent to: `return addressof(operator*());`.

```
istream_iterator& operator++();
```

3 *Requires:* `in_stream != nullptr`.

4 *Effects:* `*in_stream >> value`.

5 *Returns:* `*this`.

```
istream_iterator operator++(int);
```

6 *Requires:* `in_stream != nullptr`.

7 *Effects:*

```
istream_iterator tmp = *this;
*in_stream >> value;
return tmp;
```

```
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance> &x,
                  const istream_iterator<T, charT, traits, Distance> &y);
```

8 *Returns:* `x.in_stream == y.in_stream`.

```
template <class T, class charT, class traits, class Distance>
  bool operator==(default_sentinel x,
                  const istream_iterator<T, charT, traits, Distance> &y);
```

9 *Returns:* `nullptr == y.in_stream`.

```
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance> &x,
                  default_sentinel y);
```

10 *Returns:* `x.in_stream == nullptr`.

```
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
                  const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(default_sentinel x,
                  const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
                  default_sentinel y);
```

11 *Returns:* `!(x == y)`

21.3.8.2 Class template `ostream_iterator` [std2.ostream.iterator]

1 `ostream_iterator` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `charT*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

2 `ostream_iterator` is defined as:

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <class T, class charT = char, class traits = char_traits<charT>>
  class ostream_iterator {
  public:
    typedef ptrdiff_t difference_type;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT, traits> ostream_type;
    constexpr ostream_iterator() noexcept;
    ostream_iterator(ostream_type& s) noexcept;
    ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;
    ostream_iterator(const ostream_iterator& x) noexcept;
    ~ostream_iterator();
    ostream_iterator& operator=(const T& value);

    ostream_iterator& operator*();
    ostream_iterator& operator++();
    ostream_iterator& operator++(int);
  private:
    basic_ostream<charT, traits>* out_stream;  // exposition only
    const charT* delim;                        // exposition only
  };
}}~~}}~~
```

21.3.8.2.1 ostream_iterator constructors and destructor [std2.ostream.iterator.cons.des]

`constexpr ostream_iterator() noexcept;`

1 *Effects:* Initializes `out_stream` and `delim` with `nullptr`.

`ostream_iterator(ostream_type& s) noexcept;`

2 *Effects:* Initializes `out_stream` with `&s` and `delim` with `nullptr`.

`ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;`

3 *Effects:* Initializes `out_stream` with `&s` and `delim` with `delimiter`.

`ostream_iterator(const ostream_iterator& x) noexcept;`

4 *Effects:* Constructs a copy of `x`.

`~ostream_iterator();`

5 *Effects:* The iterator is destroyed.

21.3.8.2.2 ostream_iterator operations [std2.ostream.iterator.ops]

`ostream_iterator& operator=(const T& value);`

1 *Effects:* Equivalent to:

```
*out_stream << value;
if(delim != nullptr)
  *out_stream << delim;
return *this;
```

`ostream_iterator& operator*();`

2 *Returns:* `*this`.

```
ostream_iterator& operator++();
ostream_iterator& operator++(int);
```

3 *Returns:* `*this`.

21.3.8.3 Class template istreambuf_iterator [std2.istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (21.3.3.11) that reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the *end-of-stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(nullptr)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <class charT, class traits = char_traits<charT>>
  class istreambuf_iterator {
  public:
    typedef input_iterator_tag             iterator_category;
    typedef charT                          value_type;
    typedef typename traits::off_type      difference_type;
    typedef charT                          reference;
    typedef unspecified                    pointer;
    typedef charT                          char_type;
    typedef traits                         traits_type;
    typedef typename traits::int_type      int_type;
    typedef basic_streambuf<charT, traits> streambuf_type;
    typedef basic_istream<charT, traits>   istream_type;

    class proxy;                           // exposition only

    constexpr istreambuf_iterator() noexcept;
    constexpr istreambuf_iterator(default_sentinel) noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
    ~istreambuf_iterator() = default;
    istreambuf_iterator(istream_type& s) noexcept;
    istreambuf_iterator(streambuf_type* s) noexcept;
    istreambuf_iterator(const proxy& p) noexcept;
    charT operator*() const;
    istreambuf_iterator& operator++();
    proxy operator++(int);
    bool equal(const istreambuf_iterator& b) const;
  private:
    streambuf_type* sbuf_;                // exposition only
  };

  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT, traits>& a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
```

```
    bool operator==(default_sentinel a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT, traits>& a,
            default_sentinel b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT, traits>& a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator!=(default_sentinel a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT, traits>& a,
            default_sentinel b);
}}}}
```

21.3.8.3.1 Class template istreambuf_iterator::proxy [std2.istreambuf.iterator::proxy]

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <class charT, class traits = char_traits<charT>>
  class istreambuf_iterator<charT, traits>::proxy { // exposition only
    charT keep_;
    basic_streambuf<charT, traits>* sbuf_;
    proxy(charT c, basic_streambuf<charT, traits>* sbuf)
      : keep_(c), sbuf_(sbuf) { }
  public:
    charT operator*() { return keep_; }
  };
}}}}
```

1 Class `istreambuf_iterator<charT, traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

21.3.8.3.2 istreambuf_iterator constructors [std2.istreambuf.iterator.cons]

```
constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(default_sentinel) noexcept;
```

1 *Effects:* Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT, traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT, traits>* s) noexcept;
```

2 *Effects:* Constructs an `istreambuf_iterator` that uses the `basic_streambuf` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

```
istreambuf_iterator(const proxy& p) noexcept;
```

3 *Effects:* Constructs a `istreambuf_iterator` that uses the `basic_streambuf` object pointed to by the `proxy` object's constructor argument `p`.

21.3.8.3.3 istreambuf_iterator::operator* [std2.istreambuf.iterator::op*]

```
charT operator*() const
```

1 *Returns:* The character obtained via the `streambuf` member `sbuf_->sgetc()`.

21.3.8.3.4 `istreambuf_iterator::operator++` [std2.istreambuf.iterator::op++]

```
istreambuf_iterator&
  istreambuf_iterator<charT, traits>::operator++();
```

1 *Effects:* Equivalent to `sbuf_->sbumpc()`.

2 *Returns:* `*this`.

```
proxy istreambuf_iterator<charT, traits>::operator++(int);
```

3 *Effects:* Equivalent to: `return proxy(sbuf_->sbumpc(), sbuf_);`

21.3.8.3.5 `istreambuf_iterator::equal` [std2.istreambuf.iterator::equal]

```
bool equal(const istreambuf_iterator& b) const;
```

1 *Returns:* `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

21.3.8.3.6 `operator==` [std2.istreambuf.iterator::op==]

```
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT, traits>& a,
                  const istreambuf_iterator<charT, traits>& b);
```

1 *Effects:* Equivalent to: `return a.equal(b);`

```
template <class charT, class traits>
  bool operator==(default_sentinel a,
                  const istreambuf_iterator<charT, traits>& b);
```

2 *Effects:* Equivalent to: `return istreambuf_iterator<charT, traits>{}.equal(b);`

```
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT, traits>& a,
                  default_sentinel b);
```

3 *Effects:* Equivalent to: `return a.equal(istreambuf_iterator<charT, traits>{});`

21.3.8.3.7 `operator!=` [std2.istreambuf.iterator::op!=]

```
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT, traits>& a,
                  const istreambuf_iterator<charT, traits>& b);
template <class charT, class traits>
  bool operator!=(default_sentinel a,
                  const istreambuf_iterator<charT, traits>& b);
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT, traits>& a,
                  default_sentinel b);
```

1 *Effects:* Equivalent to: `return !(a == b);`

21.3.8.4 Class template `ostreambuf_iterator` [std2.ostreambuf.iterator]

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  template <class charT, class traits = char_traits<charT>>
  class ostreambuf_iterator {
  public:
    typedef ptrdiff_t                      difference_type;
```

```
    typedef charT                         char_type;
    typedef traits                        traits_type;
    typedef basic_streambuf<charT, traits> streambuf_type;
    typedef basic_ostream<charT, traits>   ostream_type;

    constexpr ostreambuf_iterator() noexcept;
    ostreambuf_iterator(ostream_type& s) noexcept;
    ostreambuf_iterator(streambuf_type* s) noexcept;
    ostreambuf_iterator& operator=(charT c);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);
    bool failed() const noexcept;

  private:
    streambuf_type* sbuf_;                // exposition only
  };
}}}}
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

21.3.8.4.1 `ostreambuf_iterator` constructors [std2.ostreambuf.iter.cons]

`constexpr ostreambuf_iterator() noexcept;`

1 *Effects:* Initializes `sbuf_` with `nullptr`.

`ostreambuf_iterator(ostream_type& s) noexcept;`

2 *Requires:* `s.rdbuf() != nullptr`.

3 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

`ostreambuf_iterator(streambuf_type* s) noexcept;`

4 *Requires:* `s != nullptr`.

5 *Effects:* Initializes `sbuf_` with `s`.

21.3.8.4.2 `ostreambuf_iterator` operations [std2.ostreambuf.iter.ops]

```
ostreambuf_iterator&
  operator=(charT c);
```

1 *Requires:* `sbuf_ != nullptr`.

2 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

3 *Returns:* `*this`.

`ostreambuf_iterator& operator*();`

4 *Returns:* `*this`.

```
ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);
```

5 *Returns:* `*this`.

`bool failed() const noexcept;`

6 *Requires:* `sbuf_ != nullptr`.

7 *Returns:* `true` if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

21.4 Ranges library [std2.ranges]

21.4.1 General [std2.ranges.general]

1 This ~~C~~Subclause describes components for dealing with ranges of elements.

2 The following subclauses describe range and view requirements, and components for range primitives as summarized in Table 6.

Table 6 — Ranges library summary

	Subclause	Header(s)
21.4.4	Range access	`<`~~`experimental/ranges`~~`std2/range>`
21.4.5	Range primitives	
21.4.6	Requirements	

21.4.2 decay_copy [std2.ranges.decaycopy]

[Editor's note: TODO: Replace the definition of [thread.decaycopy] with this definition, and move it where both standard libraries can make use of it. Location TBD.]

1 Several places in this ~~C~~Subclause use the expression *DECAY_COPY*`(x)`, which is expression-equivalent to:

```
decay_t<decltype((x))>(x)
```

21.4.3 Header <std2/range> synopsis [std2.range.synopsis]

```
#include <~~experimental/ranges~~std2/iterator>

namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  // 21.4.4, range access:
  ~~namespace {~~
    inline constexpr unspecified  begin = unspecified ;
    inline constexpr unspecified  end = unspecified ;
    inline constexpr unspecified  cbegin = unspecified ;
    inline constexpr unspecified  cend = unspecified ;
    inline constexpr unspecified  rbegin = unspecified ;
    inline constexpr unspecified  rend = unspecified ;
    inline constexpr unspecified  crbegin = unspecified ;
    inline constexpr unspecified  crend = unspecified ;
  ~~}~~

  // 21.4.5, range primitives:
  ~~namespace {~~
    inline constexpr unspecified  size = unspecified ;
    inline constexpr unspecified  empty = unspecified ;
    inline constexpr unspecified  data = unspecified ;
    inline constexpr unspecified  cdata = unspecified ;
  ~~}~~

  template <class T>
  using iterator_t = decltype(~~ranges~~::std2::begin(declval<T&>()));
```

```
template <class T>
using sentinel_t = decltype(ranges::std2::end(declval<T&>()));

template <class>
constexpr bool disable_sized_range = false;

template <class T>
struct enable_view { };

struct view_base { };

// 21.4.6, range requirements:

// 21.4.6.2, Range:
template <class T>
concept bool Range = see below;

// 21.4.6.3, SizedRange:
template <class T>
concept bool SizedRange = see below;

// 21.4.6.4, View:
template <class T>
concept bool View = see below;

// 21.4.6.5, @BoundedRangeCommonRange@:
template <class T>
concept bool BoundedRangeCommonRange = see below;

// 21.4.6.6, InputRange:
template <class T>
concept bool InputRange = see below;

// 21.4.6.7, OutputRange:
template <class R, class T>
concept bool OutputRange = see below;

// 21.4.6.8, ForwardRange:
template <class T>
concept bool ForwardRange = see below;

// 21.4.6.9, BidirectionalRange:
template <class T>
concept bool BidirectionalRange = see below;

// 21.4.6.10, RandomAccessRange:
template <class T>
concept bool RandomAccessRange = see below;
}}}}
```

21.4.4 Range access [std2.range.access]

1 In addition to being available via inclusion of the `<~~experimental/ranges~~std2/range>` header, the customization point objects in 21.4.4 are available when `<~~experimental/ranges~~std2/iterator>` is included.

[Editor's note: The customization point objects in this subsection all have deprecated behavior that permits them to work with rvalues. This is for compatability with the similarly named facilities in namespace `std`. The authors intend to replace the deprecated behavior with proper support for rvalue ranges, pending some ongoing work on an improved design. We hope to bring forward such a design in the summer meeting in Geneva later this year. See discussion in issue stl2#547.]

21.4.4.1 `begin` [std2.range.access.begin]

1 The name `begin` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::begin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges`~~`::std2::begin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::begin(E)` behaves similarly to `std::begin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, `(E) + 0` if `E` has array type (6.7.2).

(1.3) — Otherwise, *`DECAY_COPY`*`((E).begin())` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, *`DECAY_COPY`*`(begin(E))` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>` with overload resolution performed in a context that includes the declaration `template <class T> void begin(`~~`auto`~~`T&) = delete;` and does not include a declaration of ~~`ranges`~~`::std2::begin`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, ~~`ranges`~~`::std2::begin(E)` is ill-formed.

2 [*Note:* Whenever ~~`ranges`~~`::std2::begin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

21.4.4.2 `end` [std2.range.access.end]

1 The name `end` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::end(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges`~~`::std2::end(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::end(E)` behaves similarly to `std::end(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, `(E) + extent_v<T>`~~`::value`~~ if `E` has array type (6.7.2) `T`.

(1.3) — Otherwise, *`DECAY_COPY`*`((E).end())` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(`~~`ranges`~~`::std2::begin(E))>`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, *`DECAY_COPY`*`(end(E))` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(`~~`ranges`~~`::std2::begin(E))>` with overload resolution performed in a context that includes the declaration `template <class T> void end(`~~`auto`~~`T&) = delete;` and does not include a declaration of ~~`ranges`~~`::std2::end`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, ~~`ranges`~~`::std2::end(E)` is ill-formed.

2 [*Note:* Whenever ~~`ranges`~~`::std2::end(E)` is a valid expression, the types of ~~`ranges`~~`::std2::end(E)` and ~~`ranges`~~`::std2::begin(E)` satisfy `Sentinel`. — *end note*]

21.4.4.3 `cbegin` [std2.range.access.cbegin]

1 The name `cbegin` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::cbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to ~~`ranges`~~`::std2::begin(static_cast<const T&>(E))`.

2 Use of ~~`ranges`~~`::std2::cbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::cbegin(E)` behaves similarly to `std::cbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever ~~`ranges`~~`::std2::cbegin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

21.4.4.4 `cend` [std2.range.access.cend]

1 The name `cend` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::cend(E)` for some subexpression `E` of type `T` is expression-equivalent to ~~`ranges`~~`::std2::end(static_cast<const T&>(E))`.

2 Use of ~~`ranges`~~`::std2::cend(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::cend(E)` behaves similarly to `std::cend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever ~~`ranges`~~`::std2::cend(E)` is a valid expression, the types of ~~`ranges`~~`::std2::cend(E)` and ~~`ranges`~~`::std2::cbegin(E)` satisfy `Sentinel`. — *end note*]

21.4.4.5 `rbegin` [std2.range.access.rbegin]

1 The name `rbegin` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::rbegin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges`~~`::std2::rbegin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::rbegin(E)` behaves similarly to `std::rbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, *`DECAY_COPY`*`((E).rbegin())` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.3) — Otherwise, `make_reverse_iterator(`~~`ranges`~~`::std2::end(E))` if both ~~`ranges`~~`::std2::begin(E)` and ~~`ranges`~~`::std2::end(E)` are valid expressions of the same type `I` which meets the syntactic requirements of `BidirectionalIterator<I>` (21.3.3.14).

(1.4) — Otherwise, ~~`ranges`~~`::std2::rbegin(E)` is ill-formed.

2 [*Note:* Whenever ~~`ranges`~~`::std2::rbegin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

21.4.4.6 `rend` [std2.range.access.rend]

1 The name `rend` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::rend(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges`~~`::std2::rend(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::rend(E)` behaves similarly to `std::rend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, *`DECAY_COPY`*`((E).rend())` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(`~~`ranges`~~`::std2::rbegin(E))>`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.3) — Otherwise, make_reverse_iterator(~~ranges~~::std2::begin(E)) if both ~~ranges~~::std2::begin(E) and ~~ranges~~::std2::end(E) are valid expressions of the same type I which meets the syntactic requirements of BidirectionalIterator<I> (21.3.3.14).

(1.4) — Otherwise, ~~ranges~~::std2::rend(E) is ill-formed.

2 [*Note:* Whenever ~~ranges~~::std2::rend(E) is a valid expression, the types of ~~ranges~~::std2::rend(E) and ~~ranges~~::std2::rbegin(E) satisfy Sentinel. — *end note*]

21.4.4.7 crbegin [std2.range.access.crbegin]

1 The name crbegin denotes a customization point object (20.1.4.2.1.6). The expression ~~ranges~~::std2::crbegin(E) for some subexpression E of type T is expression-equivalent to ~~ranges~~::std2::rbegin(static_cast<const T&>(E)).

2 Use of ~~ranges~~::std2::crbegin(E) with rvalue E is deprecated. [*Note:* This deprecated usage exists so that ~~ranges~~::std2::crbegin(E) behaves similarly to std::crbegin(E) as defined in ISO/IEC 14882 when E is an rvalue. — *end note*]

3 [*Note:* Whenever ~~ranges~~::std2::crbegin(E) is a valid expression, its type satisfies Iterator. — *end note*]

21.4.4.8 crend [std2.range.access.crend]

1 The name crend denotes a customization point object (20.1.4.2.1.6). The expression ~~ranges~~::std2::crend(E) for some subexpression E of type T is expression-equivalent to ~~ranges~~::std2::rend(static_cast<const T&>(E)).

2 Use of ~~ranges~~::std2::crend(E) with rvalue E is deprecated. [*Note:* This deprecated usage exists so that ~~ranges~~::std2::crend(E) behaves similarly to std::crend(E) as defined in ISO/IEC 14882 when E is an rvalue. — *end note*]

3 [*Note:* Whenever ~~ranges~~::std2::crend(E) is a valid expression, the types of ~~ranges~~::std2::crend(E) and ~~ranges~~::std2::crbegin(E) satisfy Sentinel. — *end note*]

21.4.5 Range primitives [std2.range.primitives]

1 In addition to being available via inclusion of the <~~experimental/ranges~~std2/range> header, the customization point objects in 21.4.5 are available when <~~experimental/ranges~~std2/iterator> is included.

21.4.5.1 size [std2.range.primitives.size]

1 The name size denotes a customization point object (20.1.4.2.1.6). The expression ~~ranges~~::std2::size(E) for some subexpression E with type T is expression-equivalent to:

(1.1) — *DECAY_COPY*(extent_v<T>~~::value~~) if T is an array type (6.7.2).

(1.2) — Otherwise, *DECAY_COPY*(static_cast<const T&>(E).size()) if it is a valid expression and its type I satisfies Integral<I> and disable_sized_range<T> (21.4.6.3) is false.

(1.3) — Otherwise, *DECAY_COPY*(size(static_cast<const T&>(E))) if it is a valid expression and its type I satisfies Integral<I> with overload resolution performed in a context that includes the declaration template <class T> void size(const ~~auto~~T&) = delete; and does not include a declaration of ~~ranges~~::std2::size, and disable_sized_range<T> is false.

(1.4) — Otherwise, *DECAY_COPY*(~~ranges~~::std2::cend(E) - ~~ranges~~::std2::cbegin(E)), except that E is only evaluated once, if it is a valid expression and the types I and S of ~~ranges~~::std2::cbegin(E) and ~~ranges~~::std2::cend(E) meet the syntactic requirements of SizedSentinel<S, I> (21.3.3.10) and ForwardIterator<I>. If SizedSentinel and ForwardIterator are not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, ~~`ranges`~~`::std2::size(E)` is ill-formed.

2 [*Note:* Whenever ~~`ranges`~~`::std2::size(E)` is a valid expression, its type satisfies `Integral`. — *end note*]

21.4.5.2 `empty` [std2.range.primitives.empty]

1 The name `empty` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::empty(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `bool((E).empty())` if it is a valid expression.

(1.2) — Otherwise, ~~`ranges`~~`::std2::size(E) == 0` if it is a valid expression.

(1.3) — Otherwise, `bool(`~~`ranges`~~`::std2::begin(E) == `~~`ranges`~~`::std2::end(E))`, except that `E` is only evaluated once, if it is a valid expression and the type of ~~`ranges`~~`::std2::begin(E)` satisfies `ForwardIterator`.

(1.4) — Otherwise, ~~`ranges`~~`::std2::empty(E)` is ill-formed.

2 [*Note:* Whenever ~~`ranges`~~`::std2::empty(E)` is a valid expression, it has type `bool`. — *end note*]

21.4.5.3 `data` [std2.range.primitives.data]

1 The name `data` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::data(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges`~~`::std2::data(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::data(E)` behaves similarly to `std::data(E)` as defined in the C++ Working Paper when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, *`DECAY_COPY`*`((E).data())` if it is a valid expression of pointer to object type.

(1.3) — Otherwise, ~~`ranges`~~`::std2::begin(E)` if it is a valid expression of pointer to object type.

(1.4) — Otherwise, ~~`ranges`~~`::std2::data(E)` is ill-formed.

2 [*Note:* Whenever ~~`ranges`~~`::std2::data(E)` is a valid expression, it has pointer to object type. — *end note*]

21.4.5.4 `cdata` [std2.range.primitives.cdata]

1 The name `cdata` denotes a customization point object (20.1.4.2.1.6). The expression ~~`ranges`~~`::std2::cdata(E)` for some subexpression `E` of type `T` is expression-equivalent to ~~`ranges`~~`::std2::data(static_cast<const T&>(E))`.

2 Use of ~~`ranges`~~`::std2::cdata(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that ~~`ranges`~~`::std2::cdata(E)` has behavior consistent with ~~`ranges`~~`::std2::data(E)` when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever ~~`ranges`~~`::std2::cdata(E)` is a valid expression, it has pointer to object type. — *end note*]

21.4.6 Range requirements [std2.ranges.requirements]

21.4.6.1 General [std2.ranges.requirements.general]

1 Ranges are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. It their simplest form, a range object is one on which one can call `begin` and `end` to get an iterator (21.3.3.8) and a sentinel (21.3.3.9). To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of ranges.

2 This document defines three fundamental categories of ranges based on the syntax and semantics supported by each: *range*, *sized range* and *view*, as shown in Table 7.

Table 7 — Relations among range categories

$$\begin{tikzcd} \textbf{Sized Range} \arrow[dr] & \\ & \textbf{Range} \\ \textbf{View} \arrow[ur] & \end{tikzcd}$$

3 The `Range` concept requires only that `begin` and `end` return an iterator and a sentinel. The `SizedRange` concept refines `Range` with the requirement that the number of elements in the range can be determined in constant time using the `size` function. The `View` concept specifies requirements on a `Range` type with constant-time copy and assign operations.

4 In addition to the three fundamental range categories, this document defines a number of convenience refinements of `Range` that group together requirements that appear often in the concepts and algorithms. ~~*Bounded ranges*~~*Common ranges* are ranges for which `begin` and `end` return objects of the same type. *Random access ranges* are ranges for which `begin` returns a type that satisfies `RandomAccessIterator` (21.3.3.15). The range categories *bidirectional ranges*, *forward ranges*, *input ranges*, and *output ranges* are defined similarly.

21.4.6.2 Ranges [std2.ranges.range]

1 The `Range` concept defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and an `end` sentinel. [*Note:* Most algorithms requiring this concept simply forward to an `Iterator`-based algorithm by calling `begin` and `end`. — *end note*]

```
template <class T>
concept bool Range =
  requires(T&& t) {
    ranges::std2::begin(t); // not necessarily equality-preserving (see below)
    ranges::std2::end(t);
  };
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `Range<T>` is satisfied only if

(2.1) — `[begin(t),end(t))` denotes a range.

(2.2) — Both `begin(t)` and `end(t)` are amortized constant time and non-modifying. [*Note:* `begin(t)` and `end(t)` do not require implicit expression variations (20.3.1.1). — *end note*]

(2.3) — If `iterator_t<T>` satisfies `ForwardIterator`, `begin(t)` is equality preserving.

3 [*Note:* Equality preservation of both `begin` and `end` enables passing a `Range` whose iterator type satisfies `ForwardIterator` to multiple algorithms and making multiple passes over the range by repeated calls to `begin` and `end`. Since `begin` is not required to be equality preserving when the return type does not satisfy `ForwardIterator`, repeated calls might not return equal values or might not be well-defined; `begin` should be called at most once for such a range. — *end note*]

21.4.6.3 Sized ranges [std2.ranges.sized]

1 The `SizedRange` concept specifies the requirements of a `Range` type that knows its size in constant time with the `size` function.

```
template <class T>
concept bool SizedRange =
  Range<T> &&
  !disable_sized_range<remove_cv_t<remove_reference_t<T>>> &&
  requires(T& t) {
```

```
    { ~~ranges~~::std2::size(t) } -> ConvertibleTo<difference_type_t<iterator_t<T>>>;
  };
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `SizedRange<T>` is satisfied only if:

(2.1) — ~~ranges~~`::std2::size(t)` is $\mathcal{O}(1)$, does not modify `t`, and is equal to ~~ranges~~`::std2::distance(t)`.

(2.2) — If `iterator_t<T>` satisfies `ForwardIterator`, `size(t)` is well-defined regardless of the evaluation of `begin(t)`. [*Note:* `size(t)` is otherwise not required be well-defined after evaluating `begin(t)`. For a `SizedRange` whose iterator type does not model `ForwardIterator`, for example, `size(t)` might only be well-defined if evaluated before the first call to `begin(t)`. — *end note*]

3 [*Note:* The `disable_sized_range` predicate provides a mechanism to enable use of range types with the library that meet the syntactic requirements but do not in fact satisfy `SizedRange`. A program that instantiates a library template that requires a `Range` with such a range type `R` is ill-formed with no diagnostic required unless `disable_sized_range<remove_cv_t<remove_reference_t<R>>>` evaluates to `true` (21.1.2.1.3). — *end note*]

21.4.6.4 Views [std2.ranges.view]

1 The `View` concept specifies the requirements of a `Range` type that has constant time copy, move and assignment operators; that is, the cost of these operations is not proportional to the number of elements in the `View`.

2 [*Example:* Examples of `View`s are:

(2.1) — A `Range` type that wraps a pair of iterators.

(2.2) — A `Range` type that holds its elements by `shared_ptr` and shares ownership with all its copies.

(2.3) — A `Range` type that generates its elements on demand.

A container (20.8) is not a `View` since copying the container copies the elements, which cannot be done in constant time. — *end example*]

```
template <class T>
constexpr bool view-predicate   // exposition only
  = see below;

template <class T>
concept ~~bool~~ View =
  Range<T> &&
  Semiregular<T> &&
  view-predicate <T>;
```

3 Since the difference between `Range` and `View` is largely semantic, the two are differentiated with the help of the `enable_view` trait. Users may specialize `enable_view` to derive from `true_type` or `false_type`.

4 For a type `T`, the value of *`view-predicate`* `<T>` shall be:

(4.1) — If `enable_view<T>` has a member type `type`, `enable_view<T>::type::value`;

(4.2) — Otherwise, if `T` is derived from `view_base`, `true`;

(4.3) — Otherwise, if `T` is an instantiation of class template `initializer_list` (20.2.9), `set` (20.8.4.6), `multiset` (20.8.4.7), `unordered_set` (20.8.5.6), or `unordered_multiset` (20.8.5.7), `false`;

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

(4.5) — Otherwise, `true`.

21.4.6.5 Common ranges [std2.ranges.common]

[Editor's note: We suggest changing "`BoundedRange`" to "`CommonRange`". The authors believe this is a better name than "`ClassicRange`", which LEWG weakly preferred. The reason is that the iterator and sentinel of a Common range have the same type in *common*. A non-Common range can be turned into a Common range with the help of `common_iterator`. P0789R2 "Range Adaptors and Utilties" will be proposing a `view::common` adaptor that does precisely that.]

1 The ~~`BoundedRange`~~`CommonRange` concept specifies requirements of a `Range` type for which `begin` and `end` return objects of the same type. [*Note:* The standard containers (20.8) satisfy ~~`BoundedRange`~~`CommonRange`. — *end note*]

```
template <class T>
concept bool BoundedRangeCommonRange =
  Range<T> && Same<iterator_t<T>, sentinel_t<T>>;
```

21.4.6.6 Input ranges [std2.ranges.input]

1 The `InputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that satisfies `InputIterator` (21.3.3.11).

```
template <class T>
concept bool InputRange =
  Range<T> && InputIterator<iterator_t<T>>;
```

21.4.6.7 Output ranges [std2.ranges.output]

1 The `OutputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that satisfies `OutputIterator` (21.3.3.12).

```
template <class R, class T>
concept bool OutputRange =
  Range<R> && OutputIterator<iterator_t<R>, T>;
```

21.4.6.8 Forward ranges [std2.ranges.forward]

1 The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that satisfies `ForwardIterator` (21.3.3.13).

```
template <class T>
concept bool ForwardRange =
  InputRange<T> && ForwardIterator<iterator_t<T>>;
```

21.4.6.9 Bidirectional ranges [std2.ranges.bidirectional]

1 The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that satisfies `BidirectionalIterator` (21.3.3.14).

```
template <class T>
concept bool BidirectionalRange =
  ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;
```

21.4.6.10 Random access ranges [std2.ranges.random.access]

1 The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that satisfies `RandomAccessIterator` (21.3.3.15).

```
template <class T>
concept bool RandomAccessRange =
  BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```

[Editor's note: `dangling` moved here from the "Iterators library".]

21.4.7 Dangling wrapper [std2.dangling.wrappers]

21.4.7.1 Class template `dangling` [std2.dangling.wrap]

1 Class template `dangling` is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <CopyConstructible T>
  class dangling {
  public:
    constexpr dangling() requires DefaultConstructible<T>;
    constexpr dangling(T t);
    constexpr T get_unsafe() const;
  private:
    T value; // exposition only
  };

  template <Range R>
  using safe_iterator_t =
    conditional_t<is_lvalue_reference_v<R>::value,
      iterator_t<R>,
      dangling<iterator_t<R>>>;
}}}}
```

21.4.7.1.1 `dangling` operations [std2.dangling.wrap.ops]

21.4.7.1.1.1 `dangling` constructors [std2.dangling.wrap.op.const]

```
constexpr dangling() requires DefaultConstructible<T>;
```

1 *Effects:* Constructs a `dangling`, value-initializing `value`.

```
constexpr dangling(T t);
```

2 *Effects:* Constructs a `dangling`, initializing `value` with `t`.

21.4.7.1.1.2 `dangling::get_unsafe` [std2.dangling.wrap.op.get]

```
constexpr T get_unsafe() const;
```

1 *Returns:* `value`.

21.5 Algorithms library [std2.algorithms]

21.5.1 General [std2.algorithms.general]

1 This ~~C~~Subclause describes components that C++ programs may use to perform algorithmic operations on containers (~~C~~Subclause 20.8) and other sequences.

2 The following subclauses describe components for non-modifying sequence operations, modifying sequence operations, and sorting and related operations, as summarized in Table 8.

3 ~~To ease transition, implementations provide additional algorithm signatures that are deprecated in this document (Annex ??).~~

Header <~~experimental/ranges~~std2/algorithm> synopsis

```
#include <initializer_list>
```

Table 8 — Algorithms library summary

	Subclause	Header(s)
21.5.3	Non-modifying sequence operations	
21.5.4	Mutating sequence operations	<~~experimental/ranges~~std2/algorithm>
21.5.5	Sorting and related operations	

```
namespace ~~std { namespace experimental { namespace ranges~~std2 { inline namespace v1 {
  namespace tag {
    // 21.5.2, tag specifiers (See 21.2.4.2):
    struct in;
    struct in1;
    struct in2;
    struct out;
    struct out1;
    struct out2;
    struct fun;
    struct min;
    struct max;
    struct begin;
    struct end;
  }

  // 21.5.3, non-modifying sequence operations:
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryInvocable<projected<I, Proj>> Fun>
    tagged_pair<tag::in(I), tag::fun(Fun)>
      for_each(I first, S last, Fun f, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
```

```
    IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    for_each(Rng&& rng, Fun f, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I find(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Proj = identity,
    IndirectRelation<I2, projected<I1, Proj>> Pred = equal_to<>>
  I1
    find_end(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{}, Proj proj = Proj{});

template <ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
    IndirectRelation<iterator_t<Rng2>,
      projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  I1
    find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
                  Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, ForwardRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
```

```
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_first_of(Rng1&& rng1, Rng2&& rng2,
                  Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> Pred = equal_to<>>
  I
    adjacent_find(I first, S last, Pred pred = Pred{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng>
    adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  difference_type_t<I>
    count(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  difference_type_t<iterator_t<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  difference_type_t<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>),
              tag::in2(safe_iterator_t<Rng2>)>
    mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
```

```
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});


template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
    class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                      Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  I1
    search(I1 first1, S1 last1, I2 first2, S2 last2,
           Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  safe_iterator_t<Rng1>
    search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class T,
    class Pred = equal_to<>, class Proj = identity>
  requires IndirectlyComparable<I, const T*, Pred, Proj>
  I
    search_n(I first, S last, difference_type_t<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
  safe_iterator_t<Rng>
    search_n(Rng&& rng, difference_type_t<iterator_t<Rng>> count,
```

```
              const T& value, Pred pred = Pred{}, Proj proj = Proj{});

// 21.5.4, modifying sequence operations:
// 21.5.4.1, copy:
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy(Rng&& rng, O result);

template <InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, difference_type_t<I> n, O result);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});

template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    copy_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    copy_backward(Rng&& rng, I result);

// 21.5.4.2, move:
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    move(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyMovable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    move(Rng&& rng, O result);

template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
```

```
    move_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    move_backward(Rng&& rng, I result);

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
  requires IndirectlySwappable<I1, I2>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template <ForwardRange Rng1, ForwardRange Rng2>
  requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    swap_ranges(Rng1&& rng1, Rng2&& rng2);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    CopyConstructible F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>
  tagged_pair<tag::in(I), tag::out(O)>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
    class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<R>, Proj>)>>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    transform(Rng&& rng, O result, F op, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
    class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I1, Proj1>,
    projected<I2, Proj2>)>>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    transform(Rng1&& rng1, Rng2&& rng2, O result,
             F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
  requires Writable<I, const T2&> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  I
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});
```

```
template <InputRange Rng, class T1, class T2, class Proj = identity>
  requires Writable<iterator_t<Rng>, const T2&> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  safe_iterator_t<Rng>
    replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Writable<I, const T&>
  I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Writable<iterator_t<Rng>, const T&>
  safe_iterator_t<Rng>
    replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <class T, OutputIterator<const T&> O, Sentinel<O> S>
  O fill(O first, S last, const T& value);

template <class T, OutputRange<const T&> Rng>
  safe_iterator_t<Rng>
    fill(Rng&& rng, const T& value);
```

```
template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);

template <Iterator O, Sentinel<O> S, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate(O first, S last, F gen);

template <class Rng, CopyConstructible F>
    requires Invocable<F&> && OutputRange<Rng, result_of_t<F&()>>
  safe_iterator_t<Rng>
    generate(Rng&& rng, F gen);

template <Iterator O, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate_n(O first, difference_type_t<O> n, F gen);

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I remove(I first, S last, const T& value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity>
  requires Permutable<iterator_t<Rng>> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    remove(Rng&& rng, const T& value, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I remove_if(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
```

```
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires Permutable<I>
  I unique(I first, S last, R comp = R{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<I, O> &&
    (ForwardIterator<I> ||
     (InputIterator<O> && Same<value_type_t<I>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<I, O>)
  tagged_pair<tag::in(I), tag::out(O)>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    (ForwardIterator<iterator_t<Rng>> ||
     (InputIterator<O> && Same<value_type_t<iterator_t<Rng>>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<iterator_t<Rng>, O>)
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});

template <BidirectionalIterator I, Sentinel<I> S>
  requires Permutable<I>
  I reverse(I first, S last);

template <BidirectionalRange Rng>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    reverse(Rng&& rng);

template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);

template <BidirectionalRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    reverse_copy(Rng&& rng, O result);
```

```
template <ForwardIterator I, Sentinel<I> S>
  requires Permutable<I>
  tagged_pair<tag::begin(I), tag::end(I)>
    rotate(I first, I middle, S last);

template <ForwardRange Rng>
  requires Permutable<iterator_t<Rng>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    rotate(Rng&& rng, iterator_t<Rng> middle);

template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    rotate_copy(I first, I middle, S last, O result);

template <ForwardRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);

// 21.5.4.12, shuffle:
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  I shuffle(I first, S last, Gen&& g);

template <RandomAccessRange Rng, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  safe_iterator_t<Rng>
    shuffle(Rng&& rng, Gen&& g);

// 21.5.4.13, partitions:
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool
    is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I partition(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

```
template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});

template <BidirectionalRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
  tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
    partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O1> &&
    IndirectlyCopyable<iterator_t<Rng>, O2>
  tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
    partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});

// 21.5.5, sorting and related operations:
// 21.5.5.1, sorting:
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
```

```
  safe_iterator_t<Rng>
    stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
  I2
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                      Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, RandomAccessRange Rng2, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
      Sortable<iterator_t<Rng2>, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>>
  safe_iterator_t<Rng2>
    partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});

// 21.5.5.3, binary search:
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::begin(I), tag::end(I)>
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  bool
    binary_search(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

// 21.5.5.4, merge:
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
```

```
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    merge(Rng1&& rng1, Rng2&& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});

// 21.5.5.5, set operations:
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
  bool
    includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
```

```
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  O
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  O
    set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(I1), tag::out(O)>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 21.5.5.6, heap operations:
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
```

```
  safe_iterator_t<Rng>
    push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

// 21.5.5.7, minimum and maximum:
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(T), tag::max(T)>
    minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
              tag::max(value_type_t<iterator_t<Rng>>)>
    minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    safe_iterator_t<Rng>
      max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
    tagged_pair<tag::min(I), tag::max(I)>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    tagged_pair<tag::min(safe_iterator_t<Rng>),
                tag::max(safe_iterator_t<Rng>)>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
    bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
    bool
      lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  // 21.5.5.9, permutations:
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    bool
      next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    bool
      prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```
}}~~}}~~

4 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

5 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

6 Both in-place and copying versions are provided for certain algorithms.[4] When such a version is provided for *algorithm* it is called *algorithm_copy*. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

7 [*Note:* Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.5.14.5), or some equivalent solution. — *end note*]

8 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of `b-a` is the same as of

```
return distance(a, b);
```

9 In the description of algorithm return values, sentinel values are sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator as follows:

```
I tmp = first;
while(tmp != last)
  ++tmp;
return tmp;
```

10 Overloads of algorithms that take `Range` arguments (21.4.6.2) behave as if they are implemented by calling `begin` and `end` on the `Range` and dispatching to the overload that takes separate iterator and sentinel arguments.

11 The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

21.5.2 Tag specifiers [std2.alg.tagspec]

```
namespace tag {
  struct in { /* implementation-defined */ };
  struct in1 { /* implementation-defined */ };
  struct in2 { /* implementation-defined */ };
  struct out { /* implementation-defined */ };
  struct out1 { /* implementation-defined */ };
  struct out2 { /* implementation-defined */ };
  struct fun { /* implementation-defined */ };
  struct min { /* implementation-defined */ };
  struct max { /* implementation-defined */ };
  struct begin { /* implementation-defined */ };
```

4) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

```
  struct end { /* implementation-defined */ };
}
```

1 In the following description, let X be the name of a type in the `tag` namespace above.

2 `tag::`X is a tag specifier (21.2.4.2) such that *`TAGGET`*`(`D`, tag::`X`, `N`)` names a tagged getter (21.2.4.2) with DerivedCharacteristic D, ElementIndex N, and ElementName X.

3 [*Example:* `tag::in` is a type such that *`TAGGET`*`(`D`, tag::in, `N`)` names a type with the following interface:

```
struct __input_getter {
  constexpr decltype(auto) in() &       { return get<N>(static_cast<D&>(*this)); }
  constexpr decltype(auto) in() &&      { return get<N>(static_cast<D&&>(*this)); }
  constexpr decltype(auto) in() const & { return get<N>(static_cast<const D&>(*this)); }
};
```

— *end example*]

21.5.3 Non-modifying sequence operations [std2.alg.nonmodifying]

21.5.3.1 All of [std2.alg.all_of]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if `invoke(pred, invoke(proj, *i))` is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

21.5.3.2 Any of [std2.alg.any_of]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `false` if `[first,last)` is empty or if there is no iterator `i` in the range `[first,last)` such that `invoke(pred, invoke(proj, *i))` is `true`, and `true` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

21.5.3.3 None of [std2.alg.none_of]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
```

```
  bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if `invoke(pred, invoke(proj, *i))` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

21.5.3.4 For each [std2.alg.foreach]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryInvocable<projected<I, Proj>> Fun>
  tagged_pair<tag::in(I), tag::fun(Fun)>
    for_each(I first, S last, Fun f, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    for_each(Rng&& rng, Fun f, Proj proj = Proj{});
```

1 *Effects:* Calls `invoke(f, invoke(proj, *i))` for every iterator `i` in the range `[first,last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the result of `invoke(proj, *i)` is a mutable reference, `f` may apply nonconstant functions. — *end note*]

2 *Returns:* `{last, std::move(f)}`.

3 *Complexity:* Applies `f` and `proj` exactly `last - first` times.

4 *Remarks:* If `f` returns a result, the result is ignored.

5 [*Note:* The requirements of this algorithm are more strict than those specified in . This algorithm requires `Fun` to satisfy `CopyConstructible`, whereas the algorithm in the C++ Standard requires only `MoveConstructible`. — *end note*]

21.5.3.5 Find [std2.alg.find]

```
template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I find(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
```

```
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`, `invoke(pred, invoke(proj, *i)) == false`. Returns `last` if no such iterator is found.

2 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

21.5.3.6 Find end [std2.alg.find.end]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Proj = identity,
    IndirectRelation<I2, projected<I1, Proj>> Pred = equal_to<>>
  I1
    find_end(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{}, Proj proj = Proj{});

template <ForwardRange Rng1, ForwardRange Rng2,
    class Proj = identity,
    IndirectRelation<iterator_t<Rng2>,
      projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following condition holds: `invoke(pred, invoke(proj, *(i + n)), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate and projection.

21.5.3.7 Find first of [std2.alg.find.first.of]

```
template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  I1
    find_first_of(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, ForwardRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_first_of(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Finds an element that matches one of a set of values.

2 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following condition holds: `invoke(pred, invoke(proj1, *i), invoke(proj2, *j)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last1-first1) * (last2-first2)` applications of the corresponding predicate and the two projections.

21.5.3.8 Adjacent find [std2.alg.adjacent.find]

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> Pred = equal_to<>>
  I
    adjacent_find(I first, S last, Pred pred = Pred{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng>
    adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding condition holds: `invoke(pred, invoke(proj, *i), invoke(proj, *(i + 1))) != false`. Returns `last` if no such iterator is found.

2 *Complexity:* For a nonempty range, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value, and no more than twice as many applications of the projection.

21.5.3.9 Count [std2.alg.count]

```
template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  difference_type_t<I>
    count(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  difference_type_t<iterator_t<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  difference_type_t<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value, invoke(pred, invoke(proj, *i)) != false`.

2 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

21.5.3.10 Mismatch [std2.mismatch]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
```

```
    mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
     projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* A pair of iterators `i` and `j` such that `j == first2 + (i - first1)` and `i` is the first iterator in the range `[first1,last1)` for which the following corresponding conditions hold:

(1.1) — `j` is in the range `[first2, last2)`.

(1.2) — `*i != *(first2 + (i - first1))`

(1.3) — `!invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1))))`

Returns the pair `first1 + min(last1 - first1, last2 - first2)` and `first2 + min(last1 - first1, last2 - first2)` if such an iterator `i` is not found.

2 *Complexity:* At most `last1 - first1` applications of the corresponding predicate and both projections.

21.5.3.11 Equal [std2.alg.equal]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if for every iterator `i` in the range `[first1,last1)` the following condition holds: `invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1))))`. Otherwise, returns `false`.

2 *Complexity:* No applications of the corresponding predicate and projections if:

(2.1) — `SizedSentinel<S1, I1>` is satisfied, and

(2.2) — `SizedSentinel<S2, I2>` is satisfied, and

(2.3) — `last1 - first1 != last2 - first2`.

Otherwise, at most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate and projections.

21.5.3.12 Is permutation [std2.alg.is_permutation]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
    class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
```

```
  bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                      Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2,first2 + (last1 - first1))`, beginning with `I2 begin`, such that `equal(first1, last1, begin, pred, proj1, proj2)` returns `true` ; otherwise, returns `false`.

2 *Complexity:* No applications of the corresponding predicate and projections if:

(2.1) — `SizedSentinel<S1, I1>` is satisfied, and

(2.2) — `SizedSentinel<S2, I2>` is satisfied, and

(2.3) — `last1 - first1 != last2 - first2`.

Otherwise, exactly `last1 - first1` applications of the corresponding predicate and projections if `equal(first1, last1, first2, last2, pred, proj1, proj2)` would return `true`; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `last1 - first1`.

21.5.3.13 Search [std2.alg.search]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  I1
    search(I1 first1, S1 last1, I2 first2, S2 last2,
           Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  safe_iterator_t<Rng1>
    search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following condition holds:

```
invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) != false.
```

Returns `first1` if `[first2,last2)` is empty, otherwise returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and projections.

```
template <ForwardIterator I, Sentinel<I> S, class T,
    class Pred = equal_to<>, class Proj = identity>
  requires IndirectlyComparable<I, const T*, Pred, Proj>
```

```
  I
    search_n(I first, S last, difference_type_t<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
  safe_iterator_t<Rng>
    search_n(Rng&& rng, difference_type_t<iterator_t<Rng>> count,
             const T& value, Pred pred = Pred{}, Proj proj = Proj{});
```

4 *Effects:* Finds a subsequence of equal values in a sequence.

5 *Returns:* The first iterator `i` in the range `[first,last-count)` such that for every non-negative integer `n` less than `count` the following condition holds: `invoke(pred, invoke(proj, *(i + n)), value) != false`. Returns `last` if no such iterator is found.

6 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

21.5.4 Mutating sequence operations [std2.alg.modifying.operations]

21.5.4.1 Copy [std2.alg.copy]

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy(Rng&& rng, O result);
```

1 *Effects:* Copies elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

2 *Returns:* `{last, result + (last - first)}`.

3 *Requires:* `result` shall not be in the range `[first,last)`.

4 *Complexity:* Exactly `last - first` assignments.

```
template <InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, difference_type_t<I> n, O result);
```

5 *Effects:* For each non-negative integer $i < n$, performs `*(result + i) = *(first + i)`.

6 *Returns:* `{first + n, result + n}`.

7 *Complexity:* Exactly `n` assignments.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
```

```
template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

8 Let N be the number of iterators `i` in the range `[first,last)` for which the condition `invoke(pred, invoke(proj, *i))` holds.

9 *Requires:* The ranges `[first,last)` and `[result,result +` N`)` shall not overlap.

10 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `invoke(pred, invoke(proj, *i))` is `true`.

11 *Returns:* `{last, result +` N`}`.

12 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

13 *Remarks:* Stable (20.1.5.5.7).

```
template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    copy_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    copy_backward(Rng&& rng, I result);
```

14 *Effects:* Copies elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to `first`.[5] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

15 *Requires:* `result` shall not be in the range `(first,last]`.

16 *Returns:* `{last, result - (last - first)}`.

17 *Complexity:* Exactly `last - first` assignments.

21.5.4.2 Move [std2.alg.move]

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    move(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyMovable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    move(Rng&& rng, O result);
```

1 *Effects:* Moves elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) =` ~~`ranges`~~`::std2::iter_move(first + n)`.

2 *Returns:* `{last, result + (last - first)}`.

3 *Requires:* `result` shall not be in the range `[first,last)`.

4 *Complexity:* Exactly `last - first` move assignments.

5) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

```
template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    move_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    move_backward(Rng&& rng, I result);
```

5 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to first.[6] For each positive integer `n <= (last - first)`, performs `*(result - n) =` ~~ranges~~`::std2::iter_move(last - n)`.

6 *Requires:* `result` shall not be in the range `(first,last]`.

7 *Returns:* `{last, result - (last - first)}`.

8 *Complexity:* Exactly `last - first` assignments.

21.5.4.3 swap [std2.alg.swap]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
  requires IndirectlySwappable<I1, I2>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template <ForwardRange Rng1, ForwardRange Rng2>
  requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    swap_ranges(Rng1&& rng1, Rng2&& rng2);
```

1 *Effects:* For each non-negative integer `n < min(last1 - first1, last2 - first2)` performs: ~~ranges~~`::std2::iter_swap(first1 + n, first2 + n)`.

2 *Requires:* The two ranges `[first1,last1)` and `[first2,last2)` shall not overlap. `*(first1 + n)` shall be swappable with (20.3.3.11) `*(first2 + n)`.

3 *Returns:* `{first1 + n, first2 + n}`, where `n` is `min(last1 - first1, last2 - first2)`.

4 *Complexity:* Exactly `min(last1 - first1, last2 - first2)` swaps.

21.5.4.4 Transform [std2.alg.transform]

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    CopyConstructible F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>
  tagged_pair<tag::in(I), tag::out(O)>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
    class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<R>, Proj>)>>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    transform(Rng&& rng, O result, F op, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
```

6) `move_backward` should be used instead of move when last is in the range `[result - (last - first),result)`.

```
    WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
    class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I1, Proj1>,
    projected<I2, Proj2>)>>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
              tag::in2(safe_iterator_t<Rng2>),
              tag::out(O)>
    transform(Rng1&& rng1, Rng2&& rng2, O result,
             F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 Let N be `(last1 - first1)` for unary transforms, or `min(last1 - first1, last2 - first2) for binary transforms`.

2 *Effects:* Assigns through every iterator `i` in the range `[result,result +` N`)` a new corresponding value equal to `invoke(op, invoke(proj, *(first1 + (i - result))))` or `invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))`.

3 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,first1 +` N`]`, `[first2,first2 +` N`]`, and `[result,result +` N`]`.[7]

4 *Returns:* `{first1 +` N`, result +` N`}` or `make_tagged_tuple<tag::in1, tag::in2, tag::out>(first1 +` N`, first2 +` N`, result +` N`)`.

5 *Complexity:* Exactly N applications of `op` or `binary_op` and the corresponding projection(s).

6 *Remarks:* `result` may be equal to `first1` in case of unary transform, or to `first1` or `first2` in case of binary transform.

21.5.4.5 Replace [std2.alg.replace]

```
template <InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
  requires Writable<I, const T2&> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  I
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, class Proj = identity>
  requires Writable<iterator_t<Rng>, const T2&> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  safe_iterator_t<Rng>
    replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Writable<I, const T&>
  I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity,
```

7) The use of fully closed ranges is intentional.

```
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Writable<iterator_t<Rng>, const T&>
  safe_iterator_t<Rng>
    replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
```

1 *Effects:* Assigns `new_value` through each iterator `i` in the range `[first,last)` when the following corresponding conditions hold: `invoke(proj, *i) == old_value`, `invoke(pred, invoke(proj, *i)) != false`.

2 *Returns:* `last`.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

```
template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});
```

4 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

5 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
invoke(proj, *(first + (i - result))) == old_value
invoke(pred, invoke(proj, *(first + (i - result)))) != false
```

6 *Returns:* `{last, result + (last - first)}`.

7 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

21.5.4.6 Fill [std2.alg.fill]

```
template <class T, OutputIterator<const T&> O, Sentinel<O> S>
  O fill(O first, S last, const T& value);

template <class T, OutputRange<const T&> Rng>
  safe_iterator_t<Rng>
    fill(Rng&& rng, const T& value);

template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);
```

1 *Effects:* `fill` assigns `value` through all the iterators in the range `[first,last)`. `fill_n` assigns `value` through all the iterators in the counted range `[first,n)` if `n` is positive, otherwise it does nothing.

2 *Returns:* `last`, where `last` is `first + max(n, 0)` for `fill_n`.

3 *Complexity:* Exactly `last - first` assignments.

21.5.4.7 Generate [std2.alg.generate]

```
template <Iterator O, Sentinel<O> S, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate(O first, S last, F gen);

template <class Rng, CopyConstructible F>
    requires Invocable<F&> && OutputRange<Rng, result_of_t<F&()>>
  safe_iterator_t<Rng>
    generate(Rng&& rng, F gen);

template <Iterator O, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate_n(O first, difference_type_t<O> n, F gen);
```

1 *Effects:* The generate algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first,last)`. The `generate_n` algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the counted range `[first,n)` if `n` is positive, otherwise it does nothing.

2 *Returns:* `last`, where `last` is `first + max(n, 0)` for `generate_n`.

3 *Complexity:* Exactly `last - first` evaluations of `invoke(gen)` and assignments.

21.5.4.8 Remove [std2.alg.remove]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I remove(I first, S last, const T& value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity>
  requires Permutable<iterator_t<Rng>> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    remove(Rng&& rng, const T& value, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I remove_if(I first, S last, Pred pred, Proj proj = Proj{});
```

```
template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

2 *Returns:* The end of the resulting range.

3 *Remarks:* Stable (20.1.5.5.7).

4 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

5 *Note:* each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

6 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

7 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

8 *Returns:* A pair consisting of `last` and the end of the resulting range.

9 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

10 *Remarks:* Stable (20.1.5.5.7).

21.5.4.9 Unique [std2.alg.unique]

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> R = equal_to<>>
```

```
  requires Permutable<I>
  I unique(I first, S last, R comp = R{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
```

1 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `invoke(proj, *(i - 1)) == invoke(proj, *i)` or `invoke(pred, invoke(proj, *(i - 1)), invoke(proj, *i)) != false`.

2 *Returns:* The end of the resulting range.

3 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<I, O> &&
    (ForwardIterator<I> ||
     (InputIterator<O> && Same<value_type_t<I>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<I, O>)
  tagged_pair<tag::in(I), tag::out(O)>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    (ForwardIterator<iterator_t<Rng>> ||
     (InputIterator<O> && Same<value_type_t<iterator_t<Rng>>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<iterator_t<Rng>, O>)
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
```

4 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

5 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold:

```
invoke(proj, *i) == invoke(proj, *(i - 1))
```

or

```
invoke(pred, invoke(proj, *i), invoke(proj, *(i - 1))) != false.
```

6 *Returns:* A pair consisting of `last` and the end of the resulting range.

7 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

21.5.4.10 Reverse [std2.alg.reverse]

```
template <BidirectionalIterator I, Sentinel<I> S>
  requires Permutable<I>
  I reverse(I first, S last);
```

```
template <BidirectionalRange Rng>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    reverse(Rng&& rng);
```

1 *Effects:* For each non-negative integer `i < (last - first)/2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

2 *Returns:* `last`.

3 *Complexity:* Exactly `(last - first)/2` swaps.

```
template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);

template <BidirectionalRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    reverse_copy(Rng&& rng, O result);
```

4 *Effects:* Copies the range `[first,last)` to the range `[result,result+(last-first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

5 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

6 *Returns:* `{last, result + (last - first)}`.

7 *Complexity:* Exactly `last - first` assignments.

21.5.4.11 Rotate [std2.alg.rotate]

```
template <ForwardIterator I, Sentinel<I> S>
  requires Permutable<I>
  tagged_pair<tag::begin(I), tag::end(I)> rotate(I first, I middle, S last);

template <ForwardRange Rng>
  requires Permutable<iterator_t<Rng>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>), tag::end(safe_iterator_t<Rng>)>
    rotate(Rng&& rng, iterator_t<Rng> middle);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* `{first + (last - middle), last}`.

3 *Remarks:* This is a left rotate.

4 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges.

5 *Complexity:* At most `last - first` swaps.

```
template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    rotate_copy(I first, I middle, S last, O result);

template <ForwardRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
```

6 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

7 *Returns:* `{last, result + (last - first)}`.

8 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

9 *Complexity:* Exactly `last - first` assignments.

21.5.4.12 Shuffle [std2.alg.random.shuffle]

```
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  I shuffle(I first, S last, Gen&& g);

template <RandomAccessRange Rng, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  safe_iterator_t<Rng>
    shuffle(Rng&& rng, Gen&& g);
```

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 *Complexity:* Exactly `(last - first) - 1` swaps.

3 *Returns:* `last`

4 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

21.5.4.13 Partitions [std2.alg.partitions]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool
    is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if `[first,last)` is partitioned by `pred` and `proj`, i.e. if all iterators `i` for which `invoke(pred, invoke(proj, *i)) != false` come before those that do not, for every `i` in `[first,last)`.

2 *Complexity:* Linear. At most `last - first` applications of `pred` and `proj`.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I partition(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
```

```
    partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

3 *Effects:* Permutes the elements in the range `[first,last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

4 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

5 *Complexity:* If I meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps; otherwise at most `last - first` swaps. Exactly `last - first` applications of the predicate and projection.

```
template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});

template <BidirectionalRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

6 *Effects:* Permutes the elements in the range `[first,last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

7 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)`, `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`. The relative order of the elements in both groups is preserved.

8 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate and projection.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
  tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
    partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O1> &&
    IndirectlyCopyable<iterator_t<Rng>, O2>
  tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
    partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
```

9 *Requires:* The input range shall not overlap with either of the output ranges.

10 *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `invoke(pred, invoke(proj, *i))` is `true`, or to the output range beginning with `out_false` otherwise.

11 *Returns:* A tuple `p` such that `get<0>(p)` is `last`, `get<1>(p)` is the end of the output range beginning at `out_true`, and `get<2>(p)` is the end of the output range beginning at `out_false`.

12 *Complexity:* Exactly `last - first` applications of `pred` and `proj`.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
```

13 *Requires:* `[first,last)` shall be partitioned by `pred` and `proj`, i.e. there shall be an iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

14 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

15 *Complexity:* $\mathcal{O}(\log($`last - first`$))$ applications of `pred` and `proj`.

21.5.5 Sorting and related operations [std2.alg.sorting]

1 All the operations in 21.5.5 take an optional binary callable predicate of type `Comp` that defaults to `less<>`.

2 `Comp` is a callable object (20.5.14.3). The return value of the `invoke` operation applied to an object of type `Comp`, when contextually converted to `bool` (Clause 7), yields `true` if the first argument of the call is less than the second, and `false` otherwise. `Comp comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

3 A sequence is *sorted with respect to a comparator and projection* `comp` and `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `invoke(comp, invoke(proj, *(i + n)), invoke(proj, *i)) == false`.

4 A sequence `[start,finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < distance(start, finish)`, `f(*(start + i))` is true if and only if `i < n`.

5 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

21.5.5.1 Sorting [std2.alg.sort]

21.5.5.1.1 `sort` [std2.sort]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Returns:* `last`.

3 *Complexity:* $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons, and twice as many applications of the projection.

21.5.5.1.2 `stable_sort` [std2.stable.sort]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Returns:* `last`.

3 *Complexity:* Let N `== last - first`. If enough extra memory is available, $N \log(N)$ comparisons. Otherwise, at most $N \log^2(N)$ comparisons. In either case, twice as many applications of the projection as the number of comparisons.

4 *Remarks:* Stable (20.1.5.5.7).

21.5.5.1.3 `partial_sort` [std2.partial.sort]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});
```

1 *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

2 *Returns:* `last`.

3 *Complexity:* It takes approximately `(last - first) * log(middle - first)` comparisons, and exactly twice as many applications of the projection.

21.5.5.1.4 `partial_sort_copy` [std2.partial.sort.copy]

```
template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
  I2
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                      Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, RandomAccessRange Rng2, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
      Sortable<iterator_t<Rng2>, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>>
```

```
  safe_iterator_t<Rng2>
    partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

2 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

3 *Complexity:* Approximately

```
(last - first) * log(min(last - first, result_last - result_first))
```

comparisons, and exactly twice as many applications of the projection.

21.5.5.1.5 `is_sorted` [std2.is.sorted]

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* `is_sorted_until(first, last, comp, proj) == last`

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

2 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is sorted.

3 *Complexity:* Linear.

21.5.5.2 Nth element [std2.alg.nth.element]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});
```

1 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first,nth)` and every iterator `j` in the range `[nth,last)` it holds that: `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`.

2 *Returns:* `last`.

3 *Complexity:* Linear on average.

21.5.5.3 Binary search [std2.alg.binary.search]

1 All of the algorithms in this section are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the comparison function and projection. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

21.5.5.3.1 `lower_bound` [std2.lower.bound]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `invoke(comp, invoke(proj, e), value)`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding condition holds: `invoke(comp, invoke(proj, *j), value) != false`.

3 *Complexity:* At most $\log_2$(`last - first`) $+ \mathcal{O}(1)$ applications of the comparison function and projection.

21.5.5.3.2 `upper_bound` [std2.upper.bound]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding condition holds: `invoke(comp, value, invoke(proj, *j)) == false`.

3 *Complexity:* At most $\log_2$(`last - first`) $+ \mathcal{O}(1)$ applications of the comparison function and projection.

21.5.5.3.3 `equal_range` [std2.equal.range]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::begin(I), tag::end(I)>
```

```
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:*

```
{lower_bound(first, last, value, comp, proj),
 upper_bound(first, last, value, comp, proj)}
```

3 *Complexity:* At most $2 * \log_2(\texttt{last - first}) + \mathscr{O}(1)$ applications of the comparison function and projection.

21.5.5.3.4 `binary_search` [std2.binary.search]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  bool
    binary_search(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `invoke(comp, invoke(proj, *i), value) == false && invoke(comp, value, invoke(proj, *i)) == false`.

3 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathscr{O}(1)$ applications of the comparison function and projection.

21.5.5.4 Merge [std2.alg.merge]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = less<>,
```

```
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    merge(Rng1&& rng1, Rng2&& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`. If an element `a` precedes `b` in an input range, `a` is copied into the output range before `b`. If `e1` is an element of `[first1,last1)` and `e2` of `[first2,last2)`, `e2` is copied into the output range before `e1` if and only if `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` is `true`.

2 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `comp`, `proj1`, and `proj2`. The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result_last)`.

4 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` applications of the comparison function and each projection.

5 *Remarks:* Stable (20.1.5.5.7).

```
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});
```

6 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))` will be false.

7 *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to `comp` and `proj`.

8 *Returns:* `last`

9 *Complexity:* When enough additional memory is available, `(last - first) - 1` applications of the comparison function and projection. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

10 *Remarks:* Stable (20.1.5.5.7).

21.5.5.5 Set operations on sorted structures [std2.alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They also work with `multisets` (20.8.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multisets` in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

21.5.5.5.1 `includes` [std2.includes]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
  bool
    includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

21.5.5.5.2 `set_union` [std2.set.union]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result +` n`)`, where n is the number of elements in the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n-m, 0)$ elements from the second range shall be copied to the output range, in order.

21.5.5.5.3 `set_intersection` [std2.set.intersection]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  O
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  O
    set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

21.5.5.5.4 `set_difference` [std2.set.difference]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(I1), tag::out(O)>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `{last1, result + ` n`}`, where n is the number of elements in the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m-n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

21.5.5.5.5 `set_symmetric_difference` [std2.set.symmetric.difference]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result +` n`)`, where n is the number of elements in the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m > n$, and the last $n-m$ of these elements from `[first2,last2)` if $m < n$.

21.5.5.6 Heap operations [std2.alg.heap.operations]

1 A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key properties are:

(1) There is no element greater than `*a` in the range and

(2) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $\mathcal{O}(\log(N))$ time.

2 These properties make heaps useful as priority queues.

3 `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

21.5.5.6.1 push_heap [std2.push.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

2 *Requires:* The range `[first,last - 1)` shall be a valid heap.

3 *Returns:* `last`

4 *Complexity:* At most `log(last - first)` applications of the comparison function and projection.

21.5.5.6.2 `pop_heap` [std2.pop.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The range `[first,last)` shall be a valid non-empty heap.

2 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

3 *Returns:* `last`

4 *Complexity:* At most `2 * log(last - first)` applications of the comparison function and projection.

21.5.5.6.3 `make_heap` [std2.make.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Constructs a heap out of the range `[first,last)`.

2 *Returns:* `last`

3 *Complexity:* At most `3 * (last - first)` applications of the comparison function and projection.

21.5.5.6.4 `sort_heap` [std2.sort.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts elements in the heap `[first,last)`.

2 *Requires:* The range `[first,last)` shall be a valid heap.

3 *Returns:* `last`

4 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`), and exactly twice as many applications of the projection.

21.5.5.6.5 `is_heap` [std2.is.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* `is_heap_until(first, last, comp, proj) == last`

```
template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

2 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is a heap.

3 *Complexity:* Linear.

21.5.5.7 Minimum and maximum [std2.alg.min.max]

```
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* The smaller value.

2 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T min(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

3 *Requires:* `distance(rng) > 0`.

4 *Returns:* The smallest value in the `initializer_list` or range.

5 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

```
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

6 *Returns:* The larger value.

7 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T max(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

8 *Requires:* `distance(rng) > 0`.

9 *Returns:* The largest value in the `initializer_list` or range.

10 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

```
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

11 *Returns:* `{b, a}` if `b` is smaller than `a`, and `{a, b}` otherwise.

12 *Remarks:* Returns `{a, b}` when the arguments are equivalent.

13 *Complexity:* Exactly one comparison and exactly two applications of the projection.

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(T), tag::max(T)>
    minmax(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
              tag::max(value_type_t<iterator_t<Rng>>)>
    minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

14 *Requires:* `distance(rng) > 0`.

15 *Returns:* `{x, y}`, where `x` has the smallest and `y` has the largest value in the `initializer_list` or range.

16 *Remarks:* `x` is a copy of the leftmost argument when several arguments are equivalent to the smallest. `y` is a copy of the rightmost argument when several arguments are equivalent to the largest.

17 *Complexity:* At most `(3/2) * distance(rng)` applications of the corresponding predicate, and at most twice as many applications of the projection.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
```

```
  safe_iterator_t<Rng>
    min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

18 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`. Returns `last` if `first == last`.

19 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

20 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *i), invoke(proj, *j)) == false`. Returns `last` if `first == last`.

21 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::min(I), tag::max(I)>
    minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::min(safe_iterator_t<Rng>),
              tag::max(safe_iterator_t<Rng>)>
    minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

22 *Returns:* `{first, first}` if `[first,last)` is empty, otherwise `{m, M}`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator in `[first,last)` such that no iterator in the range refers to a larger element.

23 *Complexity:* At most $max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the comparison function and at most twice as many applications of the projection, where N is `distance(first, last)`.

21.5.5.8 Lexicographical comparison [std2.alg.lex.comparison]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
  bool
    lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
```

```
lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)` and `false` otherwise.

2 *Complexity:* At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison and projections.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
  if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
}
return first1 == last1 && first2 != last2;
```

4 *Remarks:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

21.5.5.9 Permutation generators [std2.alg.permutation.generators]

```
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>,
    class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  bool
    next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

2 *Complexity:* At most `(last - first)/2` swaps.

```
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>,
    class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  bool
    prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

3 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`.

4 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

5 *Complexity:* At most `(last - first)/2` swaps.

21.6 Numerics library [std2.numerics]

Header <~~experimental/ranges~~std2/random> synopsis

```
namespace std { namespace experimental { namespace rangesstd2 { inline namespace v1 {
  template <class G>
  concept bool UniformRandomNumberGenerator = see below;
}}}}
```

21.6.1 Uniform random number generator requirements [std2.rand.req.urng]

```
template <class G>
concept bool UniformRandomNumberGenerator =
  Invocable<G&> &&
  UnsignedIntegral<result_of_t<G&()>> &&
  requires {
    { G::min() } -> Same<result_of_t<G&()>>&&;
    { G::max() } -> Same<result_of_t<G&()>>&&;
  };
```

1 A *uniform random number generator* `g` of type `G` is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note:* The degree to which `g`'s results approximate the ideal is often determined statistically. — *end note*]

2 Let `g` be any object of type `G`. `UniformRandomNumberGenerator<G>` is satisfied only if

(2.1) — Both `G::min()` and `G::max()` are constant expressions (8.6).

(2.2) — `G::min() < G::max()`.

(2.3) — `G::min() <= g()`.

(2.4) — `g() <= G::max()`.

(2.5) — `g()` has amortized constant complexity.

Annex A (informative) Acknowledgements [acknowledgements]

This work was made possible in part by a grant from the Standard C++ Foundation, and by the support of my employer, Facebook.

Bibliography

[1] Casey Carter. Cmcstl2. https://github.com/CaseyCarter/CMCSTL2. Accessed: 2018-1-31.

[2] Casey Carter. P0898: Standard library concepts, 2 2018. http://wg21.link/P0898.

[3] Eric Niebler. Range-v3. https://github.com/ericniebler/range-v3. Accessed: 2018-1-31.

[4] Eric Niebler. P0789r1: Range adaptors and utilities, 11 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0789r1.pdf.

Index

argument, 9

C++ Standard, 2
class
  base, 9
constant iterator, 29
conventions, 8

function
  virtual member, 9

library
  Std2, 6, 9

multi-pass guarantee, 37
mutable iterator, 29

namespace, 8

projection, 3

Ranges TS, 2
requirements, 7
  iterator, 28
  uniform random number generator, 146
restriction, 9, 10

undefined behavior, 82
uniform random number generator
  requirements, 146
unspecified, 133

Index of library names

adjacent_find, 118
advance, 46
all_of, 115
any_of, 115

back_insert_iterator, 53
    back_insert_iterator, 54
    constructor, 54
back_inserter, 55
base
    counted_iterator, 73
    move_iterator, 60
    reverse_iterator, 50
bidirectional_iterator_tag, 45
BidirectionalIterator, 37
binary_search, 136

common_iterator, 65
    common_iterator, 66
    constructor, 67
    iter_move, 69
    iter_swap, 69
    operator!=, 69
    operator*, 67
    operator++, 68
    operator-, 69
    operator->, 67
    operator=, 67
    operator==, 68
copy, 121
copy_backward, 122
copy_n, 121
count, 118
    counted_iterator, 73
count_if, 118
counted_iterator, 70
    base, 73
    constructor, 72
    count, 73
    counted_iterator, 72
    iter_move, 76
    iter_swap, 76
    operator!=, 75
    operator*, 73
    operator+, 74, 76
    operator++, 73
    operator+=, 74
    operator-, 74, 76
    operator-=, 74
    operator--, 73, 74
    operator<, 75
    operator<=, 75
    operator=, 72
    operator==, 75
    operator>, 75
    operator>=, 76
    operator[], 74

dangling, 95
    dangling, 95
    get_unsafe, 95
default_sentinel, 69
difference_type, 31
difference_type_t, 31
distance, 46

empty, 45
equal, 119
    istreambuf_iterator, 84
equal_range, 135
equal_to, 12
equal_to<>, 13

failed
    ostreambuf_iterator, 85
fill, 125
fill_n, 125
find, 116
find_end, 117
find_first_of, 117
find_if, 116
find_if_not, 116
for_each, 116
forward_iterator_tag, 45
ForwardIterator, 37
front_insert_iterator, 55
    constructor, 55
    front_insert_iterator, 55
front_inserter, 56

generate, 126
generate_n, 126

get_unsafe
    dangling, 95
greater, 13
greater<>, 14
greater_equal, 13
greater_equal<>, 14

includes, 138
Incrementable, 35
indirect_result_of, 39
IndirectlyComparable, 42
IndirectlyCopyable, 41
IndirectlyCopyableStorable, 41
IndirectlyMovable, 41
IndirectlyMovableStorable, 41
IndirectlySwappable, 41
IndirectRegularUnaryInvocable, 39
IndirectRelation, 39
IndirectStrictWeakOrder, 39
IndirectUnaryInvocable, 39
IndirectUnaryPredicate, 39
inplace_merge, 137
input_iterator_tag, 45
InputIterator, 36
insert_iterator, 56
    constructor, 56
    insert_iterator, 56
inserter, 57
is_heap, 142
is_heap_until, 142
is_partitioned, 130
is_permutation, 119
is_sorted, 134
is_sorted_until, 134
istream_iterator, 78
    constructor, 79
    destructor, 79
    operator!=, 80
    operator*, 79
    operator++, 79, 80
    operator->, 79
    operator==, 80
istreambuf_iterator, 82
    constructor, 83
    operator++, 84
iter_move
    common_iterator, 69
    counted_iterator, 76
    move_iterator, 62
    reverse_iterator, 52
iter_swap
    common_iterator, 69
    counted_iterator, 76
    move_iterator, 62
    reverse_iterator, 53
Iterator, 35
iterator_category, 32
iterator_category_t, 32
iterator_traits, 43

less, 13
less<>, 14
less_equal, 13
less_equal<>, 15
lexicographical_compare, 144
lower_bound, 135

make_counted_iterator, 77
make_heap, 141
make_move_iterator, 62
make_move_sentinel, 65
make_reverse_iterator, 53
make_tagged_pair, 18
make_tagged_tuple, 19
max, 142, 143
max_element, 144
merge, 136
Mergeable, 42
min, 142
min_element, 143
minmax, 143
minmax_element, 144
mismatch, 118
movemove, 122
move_backward, 123
move_iterator, 57
    base, 60
    constructor, 59
    iter_move, 62
    iter_swap, 62
    move_iterator, 59
    operator!=, 61
    operator*, 60
    operator+, 60, 62
    operator++, 60
    operator+=, 61
    operator-, 61, 62
    operator-=, 61
    operator--, 60
    operator<, 61
    operator<=, 61
    operator=, 59

- operator==, 61
- operator>, 61
- operator>=, 62
- operator[], 61

move_sentinel, 63
- constructor, 64
- move_sentinel, 64
- operator!=, 64
- operator-, 64
- operator=, 64
- operator==, 64

next, 47
next_permutation, 145
none_of, 115
not_equal_to, 12
not_equal_to<>, 14
nth_element, 134

operator!=
- common_iterator, 69
- counted_iterator, 75
- istream_iterator, 80
- istreambuf_iterator, 84
- move_iterator, 61
- move_sentinel, 64
- reverse_iterator, 51
- unreachable, 77

operator*
- back_insert_iterator, 54
- common_iterator, 67
- counted_iterator, 73
- front_insert_iterator, 56
- insert_iterator, 57
- istream_iterator, 79
- istreambuf_iterator, 83
- move_iterator, 60
- ostream_iterator, 81
- ostreambuf_iterator, 85
- reverse_iterator, 50

operator+
- counted_iterator, 74, 76
- move_iterator, 60, 62
- reverse_iterator, 51, 52

operator++
- back_insert_iterator, 54
- common_iterator, 68
- counted_iterator, 73
- front_insert_iterator, 56
- insert_iterator, 57
- istream_iterator, 79, 80
- istreambuf_iterator, 84
- move_iterator, 60
- ostream_iterator, 82
- ostreambuf_iterator, 85
- reverse_iterator, 50

operator+=
- counted_iterator, 74
- move_iterator, 61
- reverse_iterator, 51

operator-
- common_iterator, 69
- counted_iterator, 74, 76
- move_iterator, 61, 62
- move_sentinel, 64
- reverse_iterator, 51, 52

operator-=
- counted_iterator, 74
- move_iterator, 61
- reverse_iterator, 51

operator->
- common_iterator, 67
- istream_iterator, 79
- reverse_iterator, 50

operator--
- counted_iterator, 73, 74
- move_iterator, 60
- reverse_iterator, 50

operator<
- counted_iterator, 75
- move_iterator, 61
- reverse_iterator, 52

operator<=
- counted_iterator, 75
- move_iterator, 61
- reverse_iterator, 52

operator=
- reverse_iterator, 50
- back_insert_iterator, 54
- common_iterator, 67
- counted_iterator, 72
- front_insert_iterator, 55
- insert_iterator, 57
- move_iterator, 59
- move_sentinel, 64
- ostream_iterator, 81
- ostreambuf_iterator, 85
- tagged, 17

operator==
- common_iterator, 68
- counted_iterator, 75
- istream_iterator, 80

istreambuf_iterator, 84
move_iterator, 61
move_sentinel, 64
reverse_iterator, 51
unreachable, 77
operator>
counted_iterator, 75
move_iterator, 61
reverse_iterator, 52
operator>=
counted_iterator, 76
move_iterator, 62
reverse_iterator, 52
operator[]
counted_iterator, 74
move_iterator, 61
reverse_iterator, 51
ostream_iterator, 80
constructor, 81
destructor, 81
operator*, 81
operator++, 82
operator=, 81
ostreambuf_iterator, 84
constructor, 85
output_iterator_tag, 45
OutputIterator, 37

partial_sort, 133
partial_sort_copy, 133
partition, 130
partition_copy, 131
partition_point, 132
Permutable, 42
pop_heap, 141
prev, 47
prev_permutation, 145
projected, 40
proxy
istreambuf_iterator, 83
push_heap, 140

random_access_iterator_tag, 45
RandomAccessIterator, 38
Readable, 33
remove, 126
remove_copy, 127
remove_copy_if, 127
remove_if, 126
replace, 124
replace_copy, 125
replace_copy_if, 125
replace_if, 124
reverse, 128
reverse_copy, 129
reverse_iterator, 48
reverse_iterator, 49
base, 50
constructor, 49, 50
iter_move, 52
iter_swap, 53
make_reverse_iterator non-member function, 53
operator++, 50
operator--, 50
rotate, 129
rotate_copy, 129
rvalue_reference_t, 33

search, 120
search_n, 120
Sentinel, 35
set_difference, 139
set_intersection, 138
set_symmetric_difference, 139
set_union, 138
shuffle, 130
SizedSentinel, 36
sort, 132
sort_heap, 141
Sortable, 42
stable_partition, 131
stable_sort, 133
<std2/algorithm>, 95
<std2/iterator>, 20
<std2/range>, 86
<std2/utility>, 10
swap
tagged, 18
tagged, 17
swap_ranges, 123

tagged, 15
operator=, 17
swap, 17
tagged, 16, 17
tagged_tuple
make_tagged_tuple, 19
transform, 123
tuple_element, 18
tuple_size, 18

unique, 127

unique_copy, 128
unreachable, 77
    operator!=, 77
    operator==, 77
upper_bound, 135

value_type, 32

WeaklyIncrementable, 34
Writable, 34