Merging the Ranges TS

Note: this is an early draft. It’s known to be incompleat and incorrekt, and it has lots of bad formatting.
# Contents

1 Scope

1.1 Revision History ................................................. 1

2 General Principles .................................................. 1

2.1 Goals ............................................................... 1
2.2 Rationale ........................................................... 1
2.3 Risks ............................................................... 1
2.4 Methodology ....................................................... 2
2.5 Style of presentation ............................................. 2

20 Library introduction ................................................ 4

20.1 General ............................................................. 4
20.3 Definitions ........................................................ 4
20.5 Library-wide requirements ....................................... 4

22 Concepts library .................................................... 5

22.3 Core language concepts ........................................... 5

23 General utilities library ........................................... 7

23.2 Utility components ............................................... 7
23.5 Tuples .............................................................. 8
23.6 Tagged tuple-like types .......................................... 8
23.14 Function Objects ................................................. 13

29 Ranges library ....................................................... 16

29.1 General ............................................................. 16
29.2 decay_copy ........................................................ 16
29.3 Header `<range>` synopsis ..................................... 16
29.4 Iterators library .................................................. 40
29.5 Range access ..................................................... 90
29.6 Range primitives ................................................ 92
29.7 Range requirements ............................................. 94
29.8 Dangling wrapper ............................................... 97
29.9 Algorithms library .............................................. 97

A Acknowledgements ................................................ 125

Bibliography .......................................................... 125

Index ................................................................. 126

Index of library names ............................................... 127
1 Scope

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

— Norman Maclean

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.

1.1 Revision History

1.1.1 Revision 1

— Remove section [std2.numerics] which is incorporated into P0898.
— Do not propose ranges::exchange: it is not used in the Ranges TS.
— Rewrite nearly everything to merge into std::ranges¹ rather than into std2:
  — Occurrences of "std2." in stable names are either removed, or replaced with "range" when the name resulting from removal would conflict with an existing stable name.
  — Incorporate the std2::swap customization point from P0898R0 as ranges::swap. (This was necessarily dropped from P0898R1.) Perform the necessary surgery on the Swappable concept from P0898R1 to restore the intended design that uses the renamed customization point.

2 General Principles

2.1 Goals

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.

2.2 Rationale

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]).)

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.

2.3 Risks

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.

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 ¹) std::two was another popular suggestion.
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 indicative:

(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.

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.

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

2.4 Methodology

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 §) and the common_reference type trait (ISO/IEC TS 21425:2017 §23.15.7.6) for namespace std. 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.

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 ::std::ranges.
(2.2) — The text of the Ranges TS is rebased on the latest working draft.
(2.3) — Structurally, this paper proposes to specify each piece of std::ranges alongside the content of std from the same header. Since some Ranges TS components reuse names that previously had meaning in the C++ Standard, we sometimes rename old content to avoid name collisions.
(2.4) — The content of headers from the Ranges TS with the same base name as a standard header are merged into that standard header. For example, the content of <experimental/ranges/iterator> will be merged into <iterator>. The new header <experimental/ranges/range> will be added under the name <range>.
(2.5) — 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. FIXME
(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. FIXME
(2.7) — The stable names of everything in the Ranges TS, clauses 6, 8-12 are changed by prepending “range.”. References are updated accordingly.

2.5 Style of presentation

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

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

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

In-line additions of wording from the Ranges TS to the C++ working draft are presented in cyan with underline.
In-line bits of wording that the Ranges TS strikes from the C++ working draft are presented in red with strike-through.

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

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

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

20.1 General

[Editor’s note: Insert a new row in Table 15 for the ranges library (Note that clause numbers in this table agree with the numbers in the C++ working paper for ease of review; the clauses have not been renumbered to account for the insertion of the Ranges library or the Concepts library from P0898.).]

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clause 21</td>
<td>Language support library</td>
</tr>
<tr>
<td>Clause XX</td>
<td>Concepts library</td>
</tr>
<tr>
<td>Clause 22</td>
<td>Diagnostics library</td>
</tr>
<tr>
<td>Clause 23</td>
<td>General utilities library</td>
</tr>
<tr>
<td>Clause 24</td>
<td>Strings library</td>
</tr>
<tr>
<td>Clause 25</td>
<td>Localization library</td>
</tr>
<tr>
<td>Clause 26</td>
<td>Containers library</td>
</tr>
<tr>
<td>Clause 27</td>
<td>Iterators library</td>
</tr>
<tr>
<td>Clause 28</td>
<td>Algorithms library</td>
</tr>
<tr>
<td>Clause 29</td>
<td><strong>Ranges library</strong></td>
</tr>
<tr>
<td>Clause 29</td>
<td>Numerics library</td>
</tr>
<tr>
<td>Clause 30</td>
<td>Input/output library</td>
</tr>
<tr>
<td>Clause 31</td>
<td>Regular expressions library</td>
</tr>
<tr>
<td>Clause 32</td>
<td>Atomic operations library</td>
</tr>
<tr>
<td>Clause 33</td>
<td>Thread support library</td>
</tr>
</tbody>
</table>

[Editor’s note: Modify paragraph 9 as follows:]

9 The containers (Clause 26), iterators (Clause 27), and algorithms (Clause 28), **and ranges** (Clause **Clause 29**) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

20.3 Definitions

[Editor’s note: Insert the definition of *projection* from the Ranges TS:]

20.3.18 projection

(function object argument) transformation which an algorithm applies before inspecting the values of elements

[Example:

```cpp
std::pair<int, const char*> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::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]

20.5 Library-wide requirements

20.5.1.2 Headers

[Editor’s note: Add header `<range>` to Table 16:]
Table 16 — C++ library headers

| <algorithm> | <fstream> | <new> | <string> |
| <any> | <functional> | <numeric> | <string_view> |
| <array> | <future> | <optional> | <strstream> |
| <atomic> | <initializer_list> | <ostringstream> | <sys<
| <bitset> | <iomanip> | <queue> | <system_error> |
| <charconv> | <ios> | <random> | <thread> |
| <chrono> | <iosfwd> | <range> | <tuple> |
| <codecvt> | <iostream> | <ratio> | <type_traits> |
| <compare> | <iostream> | <regex> | <typeindex> |
| <complex> | <iterator> | <scoped_allocator> | <typeinfo> |
| <concept> | <limits> | <set> | <unordered_map> |
| <condition_variable> | <list> | <shared_mutex> | <unordered_set> |
| <deque> | <locale> | <span> | <utility> |
| <exception> | <map> | <sstream> | <valarray> |
| <execution> | <memory> | <stack> | <variant> |
| <filesystem> | <memory_resource> | <std<
| <forward_list> | <mutex> | <streambuf> | <version> |

22 Concepts library  

22.3 Core language concepts  

22.3.11 Concept Swappable  

[Editor’s note: Modify the definitions of the Swappable and SwappableWith concepts as follows (This restores the Ranges TS design for these concepts from which P0898 had to deviate due to the absence of the ranges::swap customization point):]

```cpp
template <class T>
concept Swappable = is_swappable_v<T>; // see below

concept Swappable = requires(T& a, T& b) { ranges::swap(a, b); };
```

1 Let a1 and a2 denote distinct equal objects of type T, and let b1 and b2 similarly denote distinct equal objects of type T. Swappable<T> is satisfied only if:

   — After evaluating either swap(a1, b1) or swap(b1, a1) in the context described below, a1 is equal to b2 and b1 is equal to a2.

2 The context in which swap(a1, b1) or swap(b1, a1) are evaluated shall ensure that a binary non-member function named 'swap' is selected via overload resolution (16.3) on a candidate set that includes:

   — the two `swap` function templates defined in <utility> (23.2) and
   — the lookup set produced by argument-dependent lookup (6.4.2).

3 There need be no subsumption relationship between Swappable<T> and is_swappable_v<T>.

```cpp
template <class T, class U>
concept SwappableWith =
    is_swapable_with_v<T, T> && is_swapable_with_v<U, U> && // see below
    CommonReference<Const remove_reference_t<T>&, Const remove_reference_t<T>&> &&
    is_swapable_with_v<T, U> && is_swapable_with_v<U, T> && // see below
    requires(T&& t, U&& u) {
        ranges::swap(std::forward<T>(t), std::forward<T>(t));
        ranges::swap(std::forward<U>(u), std::forward<U>(u));
        ranges::swap(std::forward<T>(t), std::forward<U>(u));
        ranges::swap(std::forward<U>(u), std::forward<T>(t));
    };
```
Let $t_1$ and $t_2$ denote distinct equal objects of type `remove_cvref_t<T>`, and $E_1$ be an expression that denotes $t_1$ such that `decltype((E_1))` is $T$. Let $u_1$ and $u_2$ similarly denote distinct equal objects of type `remove_cvref_t<U>`, and $E_u$ be an expression that denotes $u_1$ such that `decltype((E_u))` is $U$. Let $C$ be `common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`.

SwappableWith<T, U> is satisfied only if:

1. After evaluating either `swap(E_t, E_u)` or `swap(E_u, E_t)` in the context described above, $C(t_1)$ is equal to $C(u_2)$ and $C(u_1)$ is equal to $C(t_2)$.

The context in which `swap(E_t, E_u)` or `swap(E_u, E_t)` are evaluated shall ensure that a binary non-member function named 'swap' is selected via overload resolution (16.3) on a candidate set that includes:

- the two `swap` function templates defined in `<utility>` (23.2) and
- the lookup set produced by argument-dependent lookup (6.4.2).

There need be no subsumption relationship between SwappableWith<T, U> and any specialization of is_swappable_with_v.

This subclause provides definitions for swappable types and expressions. In these definitions, let $t$ denote an expression of type $T$, and let $u$ denote an expression of type $U$.

An object $t$ is swappable with an object $u$ if and only if SwappableWith<T, U> is satisfied. SwappableWith<T, U> is satisfied only if given distinct objects $t_2$ equal to $t$ and $u_2$ equal to $u$, after evaluating either `ranges::swap(t, u)` or `ranges::swap(u, t)`, $t_2$ is equal to $u$ and $u_2$ is equal to $t$.

An rvalue or lvalue $t$ is swappable if and only if $t$ is swappable with any rvalue or lvalue, respectively, of type $T$.

[Example: User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

```cpp
#include <utility>

// Requires: std::forward<T>(t) shall be swappable with std::forward<U>(u).
template <class T, class SwappableWith<T, U> U>
void value_swap(T&& t, U&& u) {
    ranges::swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses "swappable with" conditions
                                                        // for rvalues and lvalues
}

// Requires: lvalues of T shall be swappable.
template <class Swappable T>
void lv_swap(T& t1, T& t2) {
    ranges::swap(t1, t2); // OK: uses swappable conditions for
                          // lvalues of type T
}

namespace N {

    struct A { int m; };
    struct Proxy { A* a; };
    Proxy proxy(A& a) { return Proxy{ &a }; }

    void swap(A& x, Proxy p) {
        ranges::swap(x.m, p.a->m); // OK: uses context equivalent to swappable
                                  // conditions for fundamental types
    }

    void swap(Proxy p, A& x) { swap(x, p); } // satisfy symmetry constraint
}

int main() {
    int i = 1, j = 2;
    lv_swap(i, j);
    assert(i == 2 && j == 1);

    N::A a1 = { 5 }, a2 = { -5 }; // N::A a1 = { 5 }, a2 = { -5 };
    value_swap(a1, proxy(a2));
    assert(a1.m == -5 && a2.m == 5);
}
```
23 General utilities library

23.2 Utility components

23.2.1 Header <utility> synopsis

[Editor’s note: Add declarations to the <utility> synopsis:

```cpp
template<size_t I>
struct in_place_index_t {
    explicit in_place_index_t() = default;
};
template<size_t I> inline constexpr in_place_index_t<I> in_place_index{};
```

```cpp
namespace experimental {
    namespace ranges {
        // 23.6.2, 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;

        // 23.6.5, tagged pairs
        template <TaggedType T1, TaggedType T2> using tagged_pair = see below;

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

        // 23.6.3, tuple-like access to tagged
        template <class Base, class... Tags>
        struct tuple_size<experimental::ranges::tagged<Base, Tags...>>;

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

        // 23.6.4, tag specifiers:
        namespace tag {
            struct in;
            struct in1;
            struct in2;
            struct out;
            struct out1;
            struct out2;
            struct fun;
            struct min;
            struct max;
            struct begin;
            struct end;
        }
    }
}
```

```cpp
namespace ranges {
    // 23.2.3, ranges::swap:
    inline namespace unspecified {
        inline constexpr unspecified swap = unspecified;
    }
}
```
The name `ranges::swap` denotes a customization point object (\(\_\_\)). The effect of the expression `ranges::swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to:

(1.1) — `(void)swap(E1, E2)`\(^2\) if that expression is valid, with overload resolution performed in a context that includes the declarations

```cpp
template <class T>
void swap(T&, T&) = delete;

template <class T, size_t N>
void swap(T(&)[N], T(&)[N]) = delete;
```

and does not include a declaration of `ranges::swap`. If the function selected by overload resolution does not exchange the values referenced by `E1` and `E2`, the program is ill-formed with no diagnostic required.

(1.2) — Otherwise, `(void)ranges::swap_ranges(E1, E2)` if `E1` and `E2` are lvalues of array types (6.7.2) with equal extent and `ranges::swap(*(E1), *(E2))` is a valid expression, except that `noexcept(ranges::swap(E1, E2))` is equal to `noexcept(ranges::swap(*(E1), *(E2)))`.

(1.3) — Otherwise, if `E1` and `E2` are lvalues of the same type `T` which meets the syntactic requirements of `MoveConstructible<T>` and `Assignable<T, T>`, exchanges the referenced values. `ranges::swap(E1, E2)` is a constant expression if the constructor selected by overload resolution for `T{std::move(E1)}` is a constexpr constructor and the expression `E1 = std::move(E2)` can appear in a constexpr function. `noexcept(ranges::swap(E1, E2))` is equal to `is_nothrow_move_constructible<T>::value && is_nothrow_move_assignable<T>::value`. If either `MoveConstructible` or `Assignable` is not satisfied, the program is ill-formed with no diagnostic required.

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

\(^2\) Remark: Whenever `ranges::swap(E1, E2)` is a valid expression, it exchanges the values referenced by `E1` and `E2` and has type `void`.

23.5 Tuples

23.5.1 Header `<tuple>` synopsis

[Editor’s note: Add declarations to `<tuple>` as follows:]

```cpp
namespace std {
    [...]  

    namespace experimental {
        namespace ranges {
            namespace v1 {
                // 23.6.6, tagged tuple:
                template <TaggedType... Types>
                    using tagged_tuple = see below;

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

[Editor’s note: Add a new subclause to Clause 23 between `[tuple]` and `[optional]`:

23.6 Tagged tuple-like types

23.6.1 General

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 23.5.3.7).
Class template tagged augments a tuple-like class type (e.g., `pair` (23.4), `tuple` (23.5)) by giving it named accessors. It is used to define the alias templates `tagged_pair` (23.6.5) and `tagged_tuple` (23.6.6).

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

```cpp
// defined in header <experimental/ranges/utility>

namespace std {
namespace experimental {
namespace ranges {
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...>, T_i, i)... {
    // see below
    using Base::Base;
    tagged() = default;
tagged(tagged&&) = default;
tagged(const tagged&) = default;
tagged(koperator=(tagged&) = default;
tagged(koperator=(const tagged&) = default;
tagged(Base&&) noexcept(see below)
        requires MoveConstructible<Base>;
tagged(const Base&) noexcept(see below)
        requires CopyConstructible<Base>;
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-remove_cvref_t<U>, tagged>
    constexpr tagged& operator=(U& u) noexcept(see below);
template <class Other>
    requires Swappable<Base>
    friend constexpr void swap(tagged&, tagged&) noexcept(see below)
        requires Swappable<Base>;
};
};
}}
```

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`.

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

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

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.

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

**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)]

```cpp
requires MoveConstructible<Base>;
```

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

**Remarks:** The expression inside **noexcept** is equivalent to:

```cpp
is_nothrow_move_constructible_v<Base> == value
```

[tagged(const Base& that) noexcept(see below)]

```cpp
requires CopyConstructible<Base>;
```

**Effects:** Initializes Base with that.

**Remarks:** The expression inside **noexcept** is equivalent to:

```cpp
is_nothrow_copy_constructible_v<Base> == value
```

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

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

**Remarks:** The expression inside **noexcept** is equivalent to:

```cpp
is_nothrow_constructible_v<Base, Other> == value
```

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

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

```cpp
tagged& operator=(tagged<Other, Tags...>&& that) noexcept(see below);
```

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

**Returns:** *this.

**Remarks:** The expression inside **noexcept** is equivalent to:

```cpp
is_nothrow_assignable_v<Base&, Other> == value
```

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

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

**Returns:** *this.
template <class U>
  requires Assignable<Base&, U> && Same<decay_remove_cvref_t<U>, tagged>
constexpr tagged& operator=(U&& u) noexcept(see below);

Effects: Assigns std::forward(u) to static_cast<Base&>(*this).
Returns: *this.
Remarks: The expression inside noexcept is equivalent to:
  is_nothrow_assignable_v<Base&, U>::value

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

Effects: Calls ranges::swap on the result of applying static_cast to *this and that.
Throws: Nothing unless the call to ranges::swap on the Base sub-objects throws.
Remarks: The expression inside noexcept is equivalent to:
  noexcept(ranges::swap(declval<Base&>(), declval<Base&>()))

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

Effects: Equivalent to lhs.swap(rhs).
Remarks: The expression inside noexcept is equivalent to:
  noexcept(lhs.swap(rhs))

23.6.3 Tuple-like access to tagged [tagged.astuple]

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

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

23.6.4 Tag specifiers [tagged.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 max { /* implementation-defined */ };
  struct begin { /* implementation-defined */ };
  struct end { /* implementation-defined */ };
}

In the following description, let X be the name of a type in the tag namespace above.

tag::X is a tag specifier (23.6.2) such that TAGGET(D, tag::X, N) names a tagged getter (23.6.2)
with DerivedCharacteristic D, ElementIndex N, and ElementName X.

[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—

23.6.5 Alias template tagged_pair

// defined in header <experimental/ranges/utility>

namespace std {
    namespace experimental {
        namespace ranges {
            inline namespace v1 {

                template <TaggedType T1, TaggedType T2>
                using tagged_pair = tagged<pair<TAGELEM(T1), TAGELEM(T2)>, TAGSPEC(T1), TAGSPEC(T2)>;

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

                —end example—

            } // v1
        } // ranges
    } // experimental
} // std

---

23.6.5.1 Tagged pair creation functions

// defined in header <experimental/ranges/utility>

namespace std {
    namespace experimental {
        namespace ranges {
            inline namespace v1 {

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

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

            } // v1
        } // ranges
    } // experimental
} // std

---

23.6.6 Alias template tagged_tuple

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

// See 23.6.4.

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—
23.6.6.1 Tagged tuple creation function

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

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]

[Editor’s note: Add declarations to <functional>:

23.14 Function Objects

23.14.1 Header <functional> synopsis

[...]

namespace ranges {

// 23.14.8, 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>;
}

[Editor’s note: Add new subclause [range.comparisons] between [comparisons] and [logical.operations]:]

23.14.8 Comparisons (ranges) [range.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 (8.5.9) 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;
   };

    operator() has effects equivalent to: return ranges::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;
   };

    operator() has effects equivalent to: return !ranges::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;
   };

    operator() has effects equivalent to: return ranges::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;
   };

    operator() has effects equivalent to: return ranges::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;
   };

    operator() has effects equivalent to: return !ranges::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;
   };

    operator() has effects equivalent to: return !ranges::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;
    using is_transparent = unspecified;
};

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.

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;
    using is_transparent = unspecified;
};

12 operator() has effects equivalent to:
   return !ranges::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;
    using is_transparent = unspecified;
};

13 operator() has effects equivalent to:
   return ranges::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;
    using is_transparent = unspecified;
};

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. For any expressions ET and EU such that decltype((ET)) is T and decltype((EU)) is U, exactly one of ranges::less<>{}(ET, EU), ranges::less<>{}(EU, ET) or ranges::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;

    using is_transparent = unspecified;
};

operator() has effects equivalent to:
    return !ranges::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;

    using is_transparent = unspecified;
};

operator() has effects equivalent to:
    return !ranges::less<>{}(std::forward<U>(u), std::forward<T>(t));

[Editor's note: Add a new clause between [algorithm] and [numerics] with the following content:]

29 Ranges library

29.1 General

This clause describes components for dealing with ranges of elements.

The following subclauses describe range and view requirements, and components for range primitives as summarized in Table 17.

Table 17 — Ranges library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.4 Iterators</td>
<td>&lt;experimental/ranges/range&gt;</td>
</tr>
<tr>
<td>29.5 Range access</td>
<td></td>
</tr>
<tr>
<td>29.6 Range primitives</td>
<td></td>
</tr>
<tr>
<td>29.7 Requirements</td>
<td></td>
</tr>
<tr>
<td>29.9 Algorithms</td>
<td></td>
</tr>
</tbody>
</table>

29.2 decay_copy

[Editor’s note: TODO: Replace the definition of [thread.decaycopy] with this definition.]

Several places in this clause use the expression `DECAY_COPY(x)`, which is expression-equivalent to:

decay_t<decltype((x))>(x)

29.3 Header <range> synopsis

```cpp
#include <experimental/ranges/range>
#include <initializer_list>

namespace std {
    namespace experimental {
        namespace ranges {
            template <class T> concept bool dereferenceable = requires(T& t) { (*t) -> auto&&; };
            // 29.4.2, iterator requirements:
            // 29.4.2.2, customization points:
```
inline namespace unspecified { 

// 29.4.2.2.1, iter_move:
inline constexpr unspecified iter_move = unspecified;

// 29.4.2.2.2, iter_swap:
inline constexpr unspecified iter_swap = unspecified;
}

// 29.4.2.3, associated types:
// 29.4.2.3.1, difference_type:
template <class> struct difference_type;
template <class T> using difference_type_t = typename difference_type<T>::type;

// 29.4.2.3.2, value_type:
template <class> struct value_type;
template <class T> using value_type_t = typename value_type<T>::type;

// 29.4.2.3.3, 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::iter_move(declval<T&>()));

// 29.4.2.4, Readable:
template <class In>
concept bool Readable = see below;

// 29.4.2.5, Writable:
template <class Out, class T>
concept bool Writable = see below;

// 29.4.2.6, WeaklyIncrementable:
template <class I>
concept bool WeaklyIncrementable = see below;

// 29.4.2.7, Incrementable:
template <class I>
concept bool Incrementable = see below;

// 29.4.2.8, Iterator:
template <class I>
concept bool Iterator = see below;

// 29.4.2.9, Sentinel:
template <class S, class I>
concept bool Sentinel = see below;

// 29.4.2.10, SizedSentinel:
template <class S, class I>
constexpr bool disable_sized_sentinel = false;

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

// 29.4.2.11, InputIterator:
template <class I>
concept bool InputIterator = see below;

// 29.4.2.12, OutputIterator:
template <class I>
concept bool OutputIterator = see below;

// 29.4.2.13, ForwardIterator:
template <class I>
concept bool ForwardIterator = see below;

// 29.4.2.14, BidirectionalIterator:
template <class I>
concept bool BidirectionalIterator = see below;

// 29.4.2.15, RandomAccessIterator:
template <class I>
concept bool RandomAccessIterator = see below;

// 29.4.3, indirect callable requirements:
// 29.4.3.2, 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 F, class... Is>
using indirect_result_of_t = typename indirect_result_of<F, Is...>::type;

// 29.4.4, common algorithm requirements:
// 29.4.4.2 IndirectlyMovable:
template <class In, class Out>
concept bool IndirectlyMovable = see below;

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

// 29.4.4.3 IndirectlyCopyable:
template <class In, class Out>
concept bool IndirectlyCopyable = see below;
template <class In, class Out>
concept bool IndirectlyCopyableStorable = see below;

// 29.4.4.4 IndirectlySwappable:
template <class I1, class I2 = I1>
concept bool IndirectlySwappable = see below;

// 29.4.4.5 IndirectlyComparable:
template <class I1, class I2, class R = equal_to<>, class P1 = identity,
class P2 = identity>
concept bool IndirectlyComparable = see below;

// 29.4.4.6 Permutable:
template <class I>
concept bool Permutable = see below;

// 29.4.4.7 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;

// 29.4.5, primitives:
// 29.4.5.1, 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>&>;

// 29.4.5.3, 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 { };

// 29.4.5.4, iterator operations:
namespace {
  constexpr unspecified advance = unspecified;
  constexpr unspecified distance = unspecified;
  constexpr unspecified next = unspecified;
  constexpr unspecified prev = unspecified;
}

template <Iterator I>
  constexpr void advance(I& i, difference_<I> n);

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

template <Iterator I, Sentinel<I> S>
  constexpr difference_<I> advance(I& i, difference_<I> n, S bound);

template <Iterator I, Sentinel<I> S>
  constexpr difference_<I> distance(I first, S last);

template <Range R>
  constexpr difference_<iterator_t<R>> distance(R&& r);

template <Iterator I>
  constexpr I next(I x);

template <Iterator I>
  constexpr I next(I x, difference_<I> n);

template <Iterator I, Sentinel<I> S>
  constexpr I next(I x, S bound);

template <Iterator I, Sentinel<I> S>
  constexpr I next(I x, difference_<I> n, S bound);
template <BidirectionalIterator I>
constexpr I prev(I x);

// 29.4.6. predefined iterators and sentinels:

// 29.4.6.1. reverse iterators:
template <BidirectionalIterator I> class reverse_iterator;

// 29.4.6.2. 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);

// 29.4.6.3, 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 <class I, Sentinel<I> 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_iterator<I>& i, const move_sentinel<S>& 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_iterator<I>& i, const move_sentinel<I>& i);
template <class I, SizedSentinel<I> S>
constexpr difference_type_t<I> operator-(const move_iterator<I>& i, const move_sentinel<I>& i);
template <class I, SizedSentinel<I> S>
constexpr difference_type_t<I> operator-(const move_sentinel<S>& i, const move_iterator<I>& i);
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);

// 29.4.6.4, 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);

// 29.4.6.5, default sentinels:
class default_sentinel;

// 29.4.6.6, 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 I1>
    constexpr bool operator==(const counted_iterator<auto>& x, default_sentinel);

template <class I1>
    constexpr bool operator!=(const counted_iterator<auto>& x, default_sentinel);
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);

// 29.4.6.7, 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;

// 29.8.1, dangling wrapper:
template <class T> class dangling;

template <Range R> using safe_iterator_t = see below;

// 29.4.7, 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!=(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!=(default_sentinel x,
    const istream_iterator<T, charT, traits, Distance>& y);

template <class T, class charT, class traits, class Distance>
bool operator==(default_sentinel x,
    default_sentinel y);

template <class T, class charT, class traits, class Distance>
bool operator!=(default_sentinel x,
    default_sentinel y);
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 ostreambuf_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;

// 29.4.5.2, iterator traits:
template <experimental::ranges::Iterator Out>
struct iterator_traits<Out>;

template <experimental::ranges::InputIterator In>
struct iterator_traits<In>  
   requires experimental::ranges::Sentinel<In, In>;

namespace ranges {

inline namespace unspecified {

   // 29.5, range access:
   inline constexpr unspecified begin = unspecified ;
   inline constexpr unspecified end = unspecified ;
   inline constexpr unspecified cbegin = unspecified ;
   inline constexpr unspecified cend = unspecified ;
   inline constexpr unspecified rbegin = unspecified ;
   inline constexpr unspecified rend = unspecified ;
   inline constexpr unspecified crbegin = unspecified ;
   inline constexpr unspecified crend = unspecified ;

   // 29.6, range primitives:
   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::begin(declval<T&>()));

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

template <class T>
struct enable_view {};

struct view_base {};

// 29.7, range requirements:

// 29.7.2, Range:
template <class T>
concept bool Range = see below;

// 29.7.3, SizedRange:
template <class T>
concept bool SizedRange = see below;

// 29.7.4, View:
template <class T>
concept bool View = see below;

// 29.7.5, BoundedRange/ CommonRange:
template <class T>
concept bool BoundedRange/ CommonRange = see below;

// 29.7.6, InputRange:
template <class T>
concept bool InputRange = see below;

// 29.7.7, OutputRange:
template <class R, class T>
concept bool OutputRange = see below;

// 29.7.8, ForwardRange:
template <class T>
concept bool ForwardRange = see below;

// 29.7.9, BidirectionalRange:
template <class T>
concept bool BidirectionalRange = see below;

// 29.7.10, RandomAccessRange:
template <class T>
concept bool RandomAccessRange = see below;

// 29.9.2, 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>
   for_each(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, const T& value, Proj proj = Proj());

template <InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
   safe_iterator_t<I>
   find_end(I first1, S 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<Rng1>, 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,
          IndirectRelation<I2, projected<I1, Proj1>> Pred = equal_to<>>
I1
   find_first_of(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<Rng1>, 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(), Proj proj = Proj());
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<>>
adjacent_find(Rng&& rng, Pred 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(Rng1), tag::in2(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>
safe_iterator_t<Rng1>
search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
  class Proj = identity>
requires IndirectlyCopyable<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{});

// 29.9.3, modifying sequence operations:
// 29.9.3.1, 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)>

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, 0 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, 0 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);

// 29.9.3.2, 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, 0 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, 0 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, 0 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& (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<I1, Proj1>,
projected<I2, Proj2>), Proj> outweighs_t<Rng1>, Proj1>,
projected<I2, 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& &&
safe_iterator_t<Rng> &&
safe_iterator_t<Rng> outweighs_t<Rng>, Proj> outweighs_t<Rng>, Proj1>,
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> outweighs_t<Rng>, Proj1>,
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>*
0 fill(0 first, S last, const T& value);

*template <class T, OutputIterator<
const T&> O>*
Rng safe_iterator_t<Rng>*fill(Rng&& rng, const T& value);

*template <class T, OutputIterator<
const T&> O>*
0 fill_n(O first, S last, const T& value);

*template <Iterator 0, Sentinel<O> S, CopyConstructible F>*
requires Invocable<
 F& && Writable<O, result_of_t<F&>()> invoke_result_t<F&>*
generate(0 first, S last, F gen);

*template <class Rng, CopyConstructible F>*
requires Invocable<
 F& && OutputRange<Rng, result_of_t<F&>()> invoke_result_t<F&>>
safe_iterator_t<Rng>*generate(Rng&& rng, F gen);

*template <Iterator 0, CopyConstructible F>*
requires Invocable<
 F& && Writable<O, result_of_t<F&>()> invoke_result_t<F&>>
O generate_n(O first, S last, const T& value, Proj proj = Proj{});

*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);

// 29.9.3.12, shuffle:
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
  UniformRandomNumberBitGenerator<remove_reference_t<Gen>> &&
  ConvertibleTo<result_of_t<Gen()>, invoke_result_t<Gen>, difference_type_t<I>>
  I shuffle(I first, S last, Gen&& g);

template <RandomAccessRange Rng, class Gen>
  requires Permutable<I> &&
  UniformRandomNumberBitGenerator<remove_reference_t<Gen>> &&
  ConvertibleTo<result_of_t<Gen()>, invoke_result_t<Gen>, difference_type_t<I>>
  safe_iterator_t<Rng>
  shuffle(Rng&& rng, Gen&& g);

// 29.9.3.13, 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 IndirectCopyable<I, O1> && IndirectCopyable<I, O2>
tagged_tuple<in(I), out1(O1), 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 IndirectCopyable<iterator_t<Rng>, O1> &&
 IndirectCopyable<iterator_t<Rng>, O2>
tagged_tuple<in(safe_iterator_t<Rng>), out1(O1), 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());

// 29.9.4, sorting and related operations:
// 29.9.4.1, 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 IndirectCopyable<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,
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,T>, 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,T>, 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{});

// 29.9.4.3, binary search:
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I,T>, Proj>> Comp = less<>>
I lower_bound(I first, S last, const T& value, Comp comp = Comp{},
             Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<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 Proj = identity,
    IndirectStrictWeakOrder<projected<I,T>, Proj>> Comp = less<>>
I upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<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 Proj = identity,
    IndirectStrictWeakOrder<projected<I,T>, 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{});

// 29.9.4.4, 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 T, 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>
safe_iterator_t<Rng>
inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{}, Proj proj = Proj{});

// 29.9.4.5, 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>, 0, 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, 0 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>, 0, Comp, Proj1, Proj2>
0 set_intersection(I1 first1, I1 last1, I2 first2, I2 last2, 0 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>, 0, Comp, Proj1, Proj2>
tagged_pair<
tag::in1(I1), tag::out(O)>
set_difference(I1 first1, I1 last1, I2 first2, I2 last2, 0 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>, 0, Comp, Proj1, Proj2>
tagged_tuple<
tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, 0 result, Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 29.9.4.6, 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{});

// 29.9.4.7, 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>>, Proj>
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(T), tag::max(T)>
minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange 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<iterator_t<Rng>, 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<iterator_t<Rng>, 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<iterator_t<Rng>, 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,
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{});

// 29.9.4.9, 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{});

29.4 Iterators library

29.4.1 General

This subclause describes components that C++ programs may use to perform iterations over containers (Clause 26), streams (30.7), and stream buffers (30.6).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 18.

Table 18 — Iterators library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.4.2 Iterator requirements</td>
<td>&lt;experimental/ranges/iterator&gt;</td>
</tr>
<tr>
<td>29.4.3 Indirect callable requirements</td>
<td></td>
</tr>
<tr>
<td>29.4.4 Common algorithm requirements</td>
<td></td>
</tr>
<tr>
<td>29.4.5 Iterator primitives</td>
<td></td>
</tr>
<tr>
<td>29.4.6 Predefined iterators</td>
<td></td>
</tr>
<tr>
<td>29.4.7 Stream iterators</td>
<td></td>
</tr>
</tbody>
</table>

29.4.2 Iterator requirements

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

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

Most of the library’s algorithmic templates that operate on data structures have interfaces that use ranges.

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

An iterator and a sentinel denoting a range are comparable. The types of a sentinel and an iterator that designate the beginning and the number of elements to which the computation is to be applied.

A range [i, n] is valid 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.

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.

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.

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 \(j == s\).

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.

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

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.

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 19.

Table 19 — Relations among iterator categories

<table>
<thead>
<tr>
<th>Random Access</th>
<th>Bidirectional</th>
<th>Forward</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>

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.

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.

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.

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.

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

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.

Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.
An invalid iterator is an iterator that may be singular.

### 29.4.2.2 Customization points

#### 29.4.2.2.1 iter_move

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

1. **(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::iter_move` but does include the lookup set produced by argument-dependent lookup (6.4.2).**
2. **(1.2) Otherwise, if the expression `*E` is well-formed:**
   1. **(1.2.1) if `*E` is an lvalue, std::move(*E);**
   2. **(1.2.2) otherwise, static_cast<decltype(*E)>(*E).**
3. **(1.3) Otherwise, `ranges::iter_move(E)` is ill-formed.**

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

#### 29.4.2.2.2 iter_swap

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

1. **(1.1) (void)iter_swap(E1, E2), if that expression is well-formed when evaluated in a context that does not include `ranges::iter_swap` but does include the lookup set produced by argument-dependent lookup (6.4.2) and the following declaration:**

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

2. **(1.2) Otherwise, if the types of `E1` and `E2` both satisfy `Readable`, and if the reference type of `E1` is swappable with () the reference type of `E2`, then `ranges::swap(*E1, *E2)`**
3. **(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.**
4. **(1.4) Otherwise, `ranges::iter_swap(E1, E2)` is ill-formed.**

If `ranges::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.

### iter_exchange_move

Iter exchange move is an exposition-only function specified as:

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

**Effects:** Equivalent to:

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

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

```cpp
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:

3) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.
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>, value_type_t<T2>>::value &&
is_nothrow_move_constructible_v<value_type_t<T1>>::value &&
noexcept(ranges::iter_move(declval<T1&>()))

29.4.2.3 Iterator associated types

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 (29.4.2.6), R is the name of a type that satisfies the Readable
concept (29.4.2.4), and II is the name of a type that satisfies the InputIterator concept (29.4.2.11) 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.

29.4.2.3.1 difference_type

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; } &&  
    requires(const T& a, const T& b) { { a - b } -> Integral; }  
    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>  
      {  
        using type = typename T::difference_type;  
      };  

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

2 Users may specialize difference_type on user-defined types.

29.4.2.3.2 value_type

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_v<T>::value, remove_cv_t<T>> { };  

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

template <class I>

43
struct value_type<\text{const } I> : value_type<\text{decay}_t\langle I\rangle> \{ \};

template <\text{class } T>
\text{requires requires \{ typename } T::\text{value_type}; \}
\text{struct value_type<T> : enable_if<is\text{object}_v\langle \text{typename } T::\text{value_type}\rangle::value, \text{typename } T::\text{value_type}> \{ \};}

template <\text{class } T>
\text{requires requires \{ typename } T::\text{element_type}; \}
\text{struct value_type<T> : enable_if<\text{is\text{object}_v\langle \text{typename } T::\text{element_type}\rangle::value, remove\text{cv}_t\langle \text{typename } T::\text{element_type}\rangle> \{ \};}

\text{template <\text{class } T> using value_type_t = \text{typename } value_type<T>::\text{type};}

2 If a type I has an associated value type, then value_type<I>::\text{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. \[\text{Note: Some legacy output iterators define a nested type named value_type that is an alias for void. These types are not Readable and have no associated value types. \text{—end note}}\]

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<\text{I::element_type}>, unless it is not an object type (6.7) in which case value_type<I> shall have no nested type type. \[\text{Note: Smart pointers like shared_ptr<int> are Readable and have an associated value type. But a smart pointer like shared_ptr<void> is not Readable and has no associated value type. \text{—end note}}\]

29.4.2.3.3 \text{iterator_category} \hspace{1cm} [\text{range.iterator.assoc.types.iterator_category}]

\text{iterator_category_t<T> is implemented as if:}

\text{template <\text{class } T> struct iterator_category \{ \};}

\text{template <\text{class } T>
\text{struct iterator_category<T*> : enable_if<is\text{object}_v\langle \text{typename } T::\text{iterator_category}\rangle::value, random\text{access}_\text{iterator}\_\text{tag}> \{ \};}

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

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

\text{template <\text{class } T> using iterator_category_t = \text{typename } iterator_category<T>::\text{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>::\text{type} is computed as follows:

1. If T::iterator_category is the same as or derives from std::random\text{access}_\text{iterator}\_\text{tag}, iterator_category<T>::\text{type} is ranges::random\text{access}_\text{iterator}\_\text{tag}.

2. Otherwise, if T::iterator_category is the same as or derives from std::bidirectional_iterator\_\text{tag}, iterator_category<T>::\text{type} is ranges::bidirectional_iterator\_\text{tag}.

44
Otherwise, if \( T \)::iterator_category is the same as or derives from `std::forward_iterator_tag`, \( \text{iterator_category} < T > \)::type is `ranges::forward_iterator_tag`.

Otherwise, if \( T \)::iterator_category is the same as or derives from `std::input_iterator_tag`, \( \text{iterator_category} < T > \)::type is `ranges::input_iterator_tag`.

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

Otherwise, \( \text{iterator_category} < T > \)::type is \( T \)::iterator_category

---

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

The expression in the requires clause is equivalent to:
```cpp
requires(T& t) { { ranges::iter_move(t) } -> auto&&; }
```

29.4.2.4 Concept Readable

The Readable concept is satisfied by types that are readable by applying \texttt{operator\*} including pointers, smart pointers, and iterators.

```cpp
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>&>;
```

29.4.2.5 Concept Writable

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

```cpp
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
    };
```

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

- If \texttt{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.
- After evaluating any above assignment expression, \( o \) is not required to be dereferenceable.
- If \( E \) is an xvalue (8.2.1), the resulting state of the object it denotes is valid but unspecified (20.5.5.15).

[Note: The only valid use of an \texttt{operator\*} is on the left side of the assignment statement. Assignment through the same value of the writable type happens only once. —end note]
29.4.2.6 Concept WeaklyIncrementable

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**.

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

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**

- The expressions \(++i\) and \(i++\) have the same domain.
- If \(i\) is incrementable, then both \(++i\) and \(i++\) advance \(i\) to the next element.
- 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]**

29.4.2.7 Concept Incrementable

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]**

\[
\text{template <class I>}
\text{concept bool Incrementable =}
\text{Regular<I> \&\&}
\text{WeaklyIncrementable<I> \&\&}
\text{requires(I i) \{}
\text{\{ i++ \} \rightarrow Same<I>\&;}
\text{\};})
\]

2 Let \(a\) and \(b\) be incrementable objects of type \(I\). **Incrementable\(<I>\) is satisfied only if**

- If \(bool(a == b)\) then \(bool(a++ == b)\).
- 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]**

29.4.2.8 Concept Iterator

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 (29.4.2.9), to read (29.4.2.11) or write (29.4.2.12) values, or to provide a richer set of iterator movements (29.4.2.13, 29.4.2.14, 29.4.2.15).
29.4.2.9 Concept Sentinel

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

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

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:

1. i == s is well-defined.
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.

29.4.2.10 Concept SizedSentinel

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.

```cpp
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>>&&;
  };
```

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:

1. If N is representable by difference_type_t<I>, then s - i is well-defined and equals N.
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 (). —end note]

4. [Note: The SizedSentinel concept is satisfied by pairs of RandomAccessIterators (29.4.2.15) and by counted iterators and their sentinels (29.4.6.6.1). —end note]

29.4.2.11 Concept InputIterator

The InputIterator concept is a refinement of Iterator (29.4.2.8). It defines requirements for a type whose referenced values can be read (from the requirement for Readable (29.4.2.4)) and which can be both pre- and post-incremented. [Note: Unlike in ISO/IEC 14882, input iterators are not required to satisfy EqualityComparable(). Unlike the input iterator requirements in 27.2.3, the InputIterator concept does not require equality comparison. —end note]

```cpp
template <class I>
concept bool InputIterator =
  Iterator<I> &&
  Readable<I> &&
  requires { typename iterator_category_t<I>; } &&
  DerivedFrom<iterator_category_t<I>, input_iterator_tag>;
```
The OutputIterator concept is a refinement of Iterator (29.4.2.8). It defines requirements for a type that can be used to write values (from the requirement for Writable (29.4.2.5)) 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
  };
```

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;
```

[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 ostream as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. —end note]

The ForwardIterator concept refines InputIterator (29.4.2.11), 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>;
```

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]

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

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

1. $a == b$ implies $++a == ++b$ and
2. The expression $\{\{X x\{++x\;\} (a), *a\} \}$ is equivalent to the expression $*a$.

[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]

The BidirectionalIterator concept refines ForwardIterator (29.4.2.13), 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> &&
  };
```
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 \( r-- \).

Let \( a \) and \( b \) be decrementable objects of type \( I \). \( \text{BidirectionalIterator}<I> \) is satisfied only if:

\[
\begin{align*}
(3.1) & \quad & \&--a == \&a. \\
(3.2) & \quad & \text{If bool}(a == b), \text{then bool}(a-- == b). \\
(3.3) & \quad & \text{If bool}(a == b), \text{then after evaluating both a-- and --b, bool}(a == b) still holds. \\
(3.4) & \quad & \text{If a is incrementable and bool}(a == b), \text{then bool}(--(++a) == b). \\
(3.5) & \quad & \text{If bool}(a == b), \text{then bool}(++(--a) == b).
\end{align*}
\]

29.4.2.15 Concept RandomAccessIterator

The \( \text{RandomAccessIterator} \) concept refines \( \text{BidirectionalIterator} \) (29.4.2.14) 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>>;
};
```

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 \( \text{difference_type_t}<I> \) such that after \( n \) applications of \( ++a \), then \( \text{bool}(a == b) \). \( \text{RandomAccessIterator}<I> \) is satisfied only if:

\[
\begin{align*}
(2.1) & \quad & (a += n) & \text{is equal to } b. \\
(2.2) & \quad & \&a += n & \text{is equal to } \&a. \\
(2.3) & \quad & (a + n) & \text{is equal to } (a += n). \\
(2.4) & \quad & \text{For any two positive integers } x \text{ and } y, \text{ if } a + (x + y) \text{ is valid, then } a + (x + y) & \text{is equal to } (a + x) + y. \\
(2.5) & \quad & a + 0 & \text{is equal to } a. \\
(2.6) & \quad & \text{If } (a + (n - 1)) \text{ is valid, then } a + n & \text{is equal to } ++(a + (n - 1)). \\
(2.7) & \quad & (b += -n) & \text{is equal to } a. \\
(2.8) & \quad & (b -= n) & \text{is equal to } a. \\
(2.9) & \quad & \&(b -= n) & \text{is equal to } \&b. \\
(2.10) & \quad & (b - n) & \text{is equal to } (b -= n). \\
(2.11) & \quad & \text{If } b \text{ is dereferenceable, then } a[n] & \text{is valid and is equal to } *b. \\
(2.12) & \quad & a <= b
\end{align*}
\]
29.4.3 Indirect callable requirements [range.indirectcallable]

29.4.3.1 General [range.indirectcallable.general]

1 There are several concepts that group requirements of algorithms that take callable objects (23.14.3) as arguments.

29.4.3.2 Indirect callables [range.indirectcallable.indirectinvocable]

1 The indirect callable concepts are used to constrain those algorithms that accept callable objects (23.14.2) as arguments.

```cpp
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>&)> invoke_result_t<F&, value_type_t<I>&>,
        result_of_t<F&(reference_t<I>&&)> invoke_result_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>&)> invoke_result_t<F&, value_type_t<I>&>,
        result_of_t<F&(reference_t<I>&&)> invoke_result_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 (Readable<Is> && ...) && Invocable<F, reference_t<Is>...>
struct indirect_result<
   (reference_t<Is>&&...)> : result
default_result<
   (reference_t<Is>&&...>)
   invoke_result<F,
   reference_t<Is>&&...> { };  

29.4.3.3 Class template projected

The `projected` class template is intended for use when specifying the constraints of algorithms that accept
callable objects and projections (20.3.18). 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`.

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

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

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

29.4.4 Common algorithm requirements

29.4.4.1 General

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 se-
quences: `IndirectlyComparable`.

[Note: The `equal_to<>` and `less<>` (23.14.8) function types used in the concepts below impose addi-
tional constraints on their arguments beyond those that appear explicitly in the concepts’ bodies. `equal_to<>` requires its arguments satisfy `EqualityComparableWith ()`, and `less<>` requires its arguments satisfy `StrictTotallyOrderedWith ()`. — end note]

29.4.4.2 Concept IndirectlyMovable

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

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

The `IndirectlyMovableStorable` concept augments `IndirectlyMovable` with additional requirements en-
abling the transfer to be performed through an intermediate object of the `Readable` type’s value type.

```cpp
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>>;
```

29.4.4.3 Concept IndirectlyCopyable

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

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

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>>;

29.4.4.4 Concept IndirectlySwappable [range.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::iter_swap(std::forward<I1>(i1), std::forward<I2>(i2));
    ranges::iter_swap(std::forward<I2>(i2), std::forward<I1>(i1));
    ranges::iter_swap(std::forward<I1>(i1), std::forward<I1>(i1));
    ranges::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::iter_swap(i1, i2), the value of *i1 is equal to the value of *i2 before the call, and vice versa.

29.4.4.5 Concept IndirectlyComparable [range.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>>;

29.4.4.6 Concept Permutable [range.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>;

29.4.4.7 Concept Mergeable [range.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>>;

29.4.4.8 Concept Sortable

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 Sortable =
Permutable<I> &&
IndirectStrictWeakOrder<R, projected<I, P>>;

29.4.5 Iterator primitives

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

29.4.5.1 Iterator traits

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:

```cpp
template <InputIterator I> struct __pointer_type {
    // exposition only
    using type = add_pointer_t<reference_t<I>>;
};
```

```cpp
template <InputIterator I>
requires requires(I i) { { i.operator->() } -> auto&&; }
struct __pointer_type<I> {
    // exposition only
    using type = decltype(declval<I>().operator->());
};
```

```cpp
template <class> struct __iterator_traits { }; // exposition only
```

```cpp
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;
};
```

```cpp
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>;
};
```

```cpp
template <class I>
using iterator_traits = __iterator_traits<I>;
```

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

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

```cpp
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]
To facilitate interoperability between new code using iterators conforming to the concepts defined in this clause and older code using iterators that conform to the iterator requirements specified in ISO/IEC 14882 27.2, three specializations of `std::iterator_traits` are provided to map the newer iterator categories and associated types to the older ones.

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

  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`.

  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`.

  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::InputIterator In>
  struct iterator_traits<In> {
  }

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

  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::reference_t<In>`.

  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`.
  
}  // namespace std
```
Let type \( C \) be `experimental::ranges::iterator_category_t<In>`. The nested type `std::iterator_traits<In>::iterator_category` is computed as follows:

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 \( C \).
2. Otherwise, if `experimental::ranges::reference_t<In>` is not a reference type, `std::iterator_traits<In>::iterator_category` is `std::input_iterator_tag`.
3. Otherwise, if \( C \) is the same as or inherits from `experimental::ranges::random_access_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `std::random_access_iterator_tag`.
4. Otherwise, if \( C \) is the same as or inherits from `experimental::ranges::bidirectional_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `std::bidirectional_iterator_tag`.
5. Otherwise, if \( C \) is the same as or inherits from `experimental::ranges::forward_iterator_tag`, `std::iterator_traits<In>::iterator_category` is `std::forward_iterator_tag`.
6. Otherwise, `std::iterator_traits<In>::iterator_category` is `std::input_iterator_tag`.

[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 27.2. —end note]

29.4.5.3 Standard iterator tags

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.

```cpp
namespace std::ranges { namespace experimental { namespace ranges { 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 { }; 
}}}}
```

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

[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:

```cpp
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]
29.4.5.4 Iterator operations

Since only types that satisfy RandomAccessIterator provide the + operator, and types that satisfy Sized-Sentinel provide the - operator, the library provides customization point objects function templates advance, distance, next, and prev. These customization point objects function templates 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.

The function templates defined in this subclause are not found by argument-dependent name lookup (6.4.2).

When found by unqualified (6.4.1) name lookup for the postfix-expression in a function call (8.5.1.2), they inhibit argument-dependent name lookup.

Example:

```cpp
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1,2,3};
    distance(begin(vec), end(vec)); // #1
}
```

The function call expression at #1 invokes std::ranges::distance, not std::distance, despite that (a) the iterator type returned from begin(vec) and end(vec) may be associated with namespace std and (b) std::distance is more specialized (17.5.6.2) than std::ranges::distance since the former requires its first two parameters to have the same type. — end example]

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

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

Requires: n shall be negative only for bidirectional iterators.

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() advance(I& i, S bound) const;

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

Effects:

(7.1) — If Assignable<I&, S> is satisfied, equivalent to i = std::move(bound).
(7.2) — Otherwise, if SizedSentinel<S, I> is satisfied, equivalent to advance(i, bound - i).
(7.3) — Otherwise, increments i until i == bound.

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

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.

Effects:

(9.1) — If SizedSentinel<S, I> is satisfied:
(9.1.1) — If |n| >= |bound - i|, equivalent to advance(i, bound).
(9.1.2) — Otherwise, equivalent to advance(i, n).
(9.2) — Otherwise, increments (or decrements for negative n) iterator i either n times or until i == bound, whichever comes first.

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

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() distance(I first, S last) const;
Requires: \([\text{first, last})\] shall denote a range, or \((\text{Same<}\text{S, I}> && \text{SizedSentinel<}\text{S, I}>))\) shall be satisfied and \([\text{last, first})\] shall denote a range.

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

\[
\begin{align*}
\text{template <Range R>} & \\
& \text{constexpr difference_type_t<iterator_t<R>> operator() distance(R& r) const;}
\end{align*}
\]

Effects: If \text{SizedRange<R>} is satisfied, equivalent to:
\[
\text{return ranges::size(r); // 29.6.1}
\]

Otherwise, equivalent to:
\[
\text{return distance(ranges::begin(r), ranges::end(r)); // 29.5}
\]

Remarks: Instantiations of this member function template may be ill-formed if the declarations in \text{<experimental/ranges/range>} are not in scope at the point of instantiation (17.6.4.1).

\[
\begin{align*}
\text{template <SizedRange R>} & \\
& \text{constexpr difference_type_t<iterator_t<R>> operator()(R&& r) const;}
\end{align*}
\]

Effects: Equivalent to:
\[
\text{return ranges::size(r); (29.6.1)}
\]

Remarks: Instantiations of this member function template may be ill-formed if the declarations in \text{<experimental/ranges/range>} are not in scope at the point of instantiation (17.6.4.1).

The name \text{next} denotes a customization point object. It has the following function call operators:

\[
\begin{align*}
\text{template <Iterator I>} & \\
& \text{constexpr I operator() next(I x) const;}
\end{align*}
\]

Effects: Equivalent to: \(+x\); return \(x\);

\[
\begin{align*}
\text{template <Iterator I>} & \\
& \text{constexpr I operator() next(I x, difference_type_t<I> n) const;}
\end{align*}
\]

Effects: Equivalent to: \(\text{advance(x, n)}\); return \(x\);

\[
\begin{align*}
\text{template <Iterator I, Sentinel<I> S>} & \\
& \text{constexpr I operator() next(I x, S bound) const;}
\end{align*}
\]

Effects: Equivalent to: \(\text{advance(x, bound)}\); return \(x\);

\[
\begin{align*}
\text{template <Iterator I, Sentinel<I> S>} & \\
& \text{constexpr I operator() next(I x, difference_type_t<I> n, S bound) const;}
\end{align*}
\]

Effects: Equivalent to: \(\text{advance(x, n, bound)}\); return \(x\);

The name \text{prev} denotes a customization point object. It has the following function call operators:

\[
\begin{align*}
\text{template <BidirectionalIterator I>} & \\
& \text{constexpr I operator() prev(I x) const;}
\end{align*}
\]

Effects: Equivalent to: \(-x\); return \(x\);

\[
\begin{align*}
\text{template <BidirectionalIterator I>} & \\
& \text{constexpr I operator() prev(I x, difference_type_t<I> n) const;}
\end{align*}
\]

Effects: Equivalent to: \(\text{advance(x, -n)}\); return \(x\);

\[
\begin{align*}
\text{template <BidirectionalIterator I>} & \\
& \text{constexpr I operator() prev(I x, difference_type_t<I> n, I bound) const;}
\end{align*}
\]

Effects: Equivalent to: \(\text{advance(x, -n, bound)}\); return \(x\);
29.4.6 Iterator adaptors

29.4.6.1 Reverse iterators

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)`.

29.4.6.1.1 Class template `reverse_iterator`

```cpp
namespace std::ranges { namespace experimental { namespace ranges { 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<U>& i);
      template <ConvertibleTo<I> U> constexpr reverse_iterator& operator=(const reverse_iterator<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;
      template <IndirectlySwappable<I> I2> friend constexpr void iter_swap(const reverse_iterator& x, const reverse_iterator<I2>& y) noexcept;

    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>
  }
}}
```
29.4.6.1.2 reverse_iterator operations

29.4.6.1.2.1 reverse_iterator constructor

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>> i);

3 Effects: Initializes current with i.current.

29.4.6.1.2.2 reverse_iterator::operator=

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

1 Effects: Assigns i.current to current.

2 Returns: *this.
29.4.6.1.2.3 Conversion

constexpr I base() const;

Returns: current.

29.4.6.1.2.4 operator*

constexpr reference operator*() const;

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

29.4.6.1.2.5 operator->

constexpr pointer operator->() const;

Effects: Equivalent to: return prev(current);

29.4.6.1.2.6 operator++

constexpr reverse_iterator& operator++();

Effects: --current;

Returns: *this.

29.4.6.1.2.7 operator--

constexpr reverse_iterator& operator--();

Effects: ++current

Returns: *this.

29.4.6.1.2.8 operator+

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

Returns: reverse_iterator(current-n).

29.4.6.1.2.9 operator+=

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

Effects: current -= n;

Returns: *this.

29.4.6.1.2.10 operator-

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

Returns: reverse_iterator(current+n).
29.4.6.1.2.11 operator-=

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

1 Effects: current += n;
2 Returns: *this.

29.4.6.1.2.12 operator[]

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

1 Returns: current[-n-1].

29.4.6.1.2.13 operator==

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;

29.4.6.1.2.14 operator!=

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;

29.4.6.1.2.15 operator<

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;

29.4.6.1.2.16 operator>

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;

29.4.6.1.2.17 operator>=

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;
29.4.6.1.2.18  operator<=

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

**Effects:** Equivalent to: \( \text{return } x.\text{current } \geq y.\text{current} \);

29.4.6.1.2.19  operator-

```cpp
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);
```

**Effects:** Equivalent to: \( \text{return } y.\text{current } - x.\text{current} \);

29.4.6.1.2.20  operator+

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

**Effects:** Equivalent to: \( \text{return } \text{reverse}_\text{iterator}<I>(x.\text{current } - n) \);

29.4.6.1.2.21  iter_move

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

**Effects:** Equivalent to: \( \text{return } \text{ranges::iter}_\text{move}(\text{prev}(i.\text{current})) \);

**Remarks:** The expression in \text{noexcept} is equivalent to:

\[
\text{noexcept(ranges::iter}_\text{move(declval<I&>())) }\&\& \text{noexcept(}--\text{declval<I&>()) }\&\&
\text{is_nothrow_copy_constructible}_\text{v}<I>\_\text{value}
\]

29.4.6.1.2.22  iter_swap

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

**Effects:** Equivalent to \( \text{ranges::iter}_\text{swap}(\text{prev}(x.\text{current}), \text{prev}(y.\text{current})) \).

**Remarks:** The expression in \text{noexcept} is equivalent to:

\[
\text{noexcept(ranges::iter}_\text{swap(declval<I>()}, \text{declval<I>())) }\&\& \text{noexcept(}--\text{declval}<I&()>)
\]

29.4.6.1.2.23  Non-member function make_reverse_iterator()

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

**Returns:** \( \text{reverse}_\text{iterator}<I>(i) \).

29.4.6.2  Insert iterators

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,

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

causes a range \([\text{first}, \text{last})\) 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.
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.

### 29.4.6.2.1 Class template back_insert_iterator

```cpp
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
};
```

### 29.4.6.2.2 back_insert_iterator operations

#### 29.4.6.2.2.1 back_insert_iterator constructor

```cpp
constexpr back_insert_iterator();

explicit back_insert_iterator(Container& x);
```

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

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

**Effects:** Initializes `container` with `&x`.

#### 29.4.6.2.2.2 back_insert_iterator::operator=

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

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

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

**Effects:** Equivalent to `container->push_back(std::move(value))`.  
**Returns:** *this.
29.4.6.2.3 back_insert_iterator::operator*  
back_insert_iterator& operator*();  
Returns: *this.

29.4.6.2.4 back_insert_iterator::operator++  
back_insert_iterator& operator++();  
back_insert_iterator operator++(int);  
Returns: *this.

29.4.6.2.5 back_inserter  
template <class Container>  
back_insert_iterator<Container> back_inserter(Container& x);  
Returns: back_insert_iterator<Container>(x).

29.4.6.2.3 Class template front_insert_iterator  
namespace std::ranges { namespace experimental { namespace ranges { inline namespace v1 {

namespace std::ranges { namespace experimental { namespace ranges { inline namespace v1 {

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);

29.4.6.2.4 front_insert_iterator operations  
29.4.6.2.4.1 front_insert_iterator constructor  
constexpr front_insert_iterator();  
Effects: Value-initializes container.

explicit front_insert_iterator(Container& x);  
Effects: Initializes container with addressof(x).

29.4.6.2.4.2 front_insert_iterator::operator=  
front_insert_iterator&  
operator=(const value_type_t<Container>& value);  
Effects: Equivalent to container->push_front(value).

Returns: *this.

front_insert_iterator&  
operator=(value_type_t<Container>&& value);
Effects: Equivalent to container->push_front(std::move(value)).

Returns: *this.

29.4.6.2.4.3 front_insert_iterator::operator*
front_insert_iterator& operator*();
Returns: *this.

29.4.6.2.4.4 front_insert_iterator::operator++
front_insert_iterator& operator++();
front_insert_iterator operator++(int);
Returns: *this.

29.4.6.2.4.5 front_inserter

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

29.4.6.2.5 Class template insert_iterator

namespace std::ranges { namespace experimental { namespace ranges { inline namespace v1 {

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);
}
}}

29.4.6.2.6 insert_iterator operations

29.4.6.2.6.1 insert_iterator constructor
insert_iterator();
Effects: Value-initializes container and iter.
insert_iterator(Container& x, iterator_t<Container> i);
Requirements: i is an iterator into x.
Effects: Initializes container with addressof(x) and iter with i.
29.4.6.2.6.2 insert_iterator::operator=

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

1 \textit{Effects:} Equivalent to:
\begin{itemize}
  \item \texttt{iter = container->insert(iter, value)};
  \item \texttt{++iter};
\end{itemize}

2 \textit{Returns:} \texttt{*this}.

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

3 \textit{Effects:} Equivalent to:
\begin{itemize}
  \item \texttt{iter = container->insert(iter, std::move(value))};
  \item \texttt{++iter};
\end{itemize}

4 \textit{Returns:} \texttt{*this}.

29.4.6.2.6.3 insert_iterator::operator*

insert_iterator& operator*();

1 \textit{Returns:} \texttt{*this}.

29.4.6.2.6.4 insert_iterator::operator++

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

1 \textit{Returns:} \texttt{*this}.

29.4.6.3 Move iterators and sentinels

29.4.6.3.1 Class template move_iterator

Class template \texttt{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 \textit{Example:}

\begin{verbatim}
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{verbatim}

namespace std::ranges { namespace experimental { namespace ranges { 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();

    } // class move_iterator

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

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

constexpr I base() const;
constexpr reference operator*() const;
constexpr move_iterator& operator=(const move_iterator<ConvertibleTo<I> U>& i);

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

67
```cpp
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

29.4.6.3.2 move_iterator operations

29.4.6.3.2.1 move_iterator constructors

constexpr move_iterator();
  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);
  Effects: Constructs a move_iterator, initializing current with i.

template <ConvertibleTo<I> U>
  constexpr move_iterator(const move_iterator<ConvertibleTo<I> U>& i);
  Effects: Constructs a move_iterator, initializing current with i.current.

29.4.6.3.2.2 move_iterator::operator=

template <ConvertibleTo<I> U>
  constexpr move_iterator& operator=(const move_iterator<ConvertibleTo<I> U>& i);
  Effects: Assigns i.current to current.

29.4.6.3.2.3 move_iterator conversion

constexpr I base() const;
  Returns: current.

29.4.6.3.2.4 move_iterator::operator*

constexpr reference operator*() const;
  Effects: Equivalent to: return iter_move(current);

29.4.6.3.2.5 move_iterator::operator++

constexpr move_iterator& operator++();
  Effects: Equivalent to ++current.

  Returns: *this.

constexpr void operator++(int);
  Effects: Equivalent to ++current.

constexpr move_iterator operator++(int)
  requires ForwardIterator<I>;
  Effects: Equivalent to:
    move_iterator tmp = *this;
    ++current;
    return tmp;
```
29.4.6.3.2.6  move_iterator::operator--

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;

29.4.6.3.2.7  move_iterator::operator+

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

1  Effects: Equivalent to: return move_iterator(current + n);

29.4.6.3.2.8  move_iterator::operator+=

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

1  Effects: Equivalent to: current += n.
2  Returns: *this.

29.4.6.3.2.9  move_iterator::operator-

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

1  Effects: Equivalent to: return move_iterator(current - n);

29.4.6.3.2.10  move_iterator::operator-=

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

1  Effects: Equivalent to: current -= n.
2  Returns: *this.

29.4.6.3.2.11  move_iterator::operator[]

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

1  Effects: Equivalent to: return iter_move(current + n);

29.4.6.3.2.12  move_iterator comparisons

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

29.4.6.3.2.13 move_iterator non-member functions [range.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);
  
  // 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);
  
  // Effects: Equivalent to: return x + n;

friend constexpr rvalue_reference_t<I> iter_move(const move_iterator& i)
  noexcept(see below);
  
  // Effects: Equivalent to: return ranges::iter_move(i.current);
  
  // Remarks: The expression in noexcept is equivalent to:
  //           noexcept(ranges::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);
  
  // Effects: Equivalent to: ranges::iter_swap(x.current, y.current).
  
  // Remarks: The expression in noexcept is equivalent to:
  //           noexcept(ranges::iter_swap(x.current, y.current))

template <InputIterator I>
  constexpr move_iterator<I> make_move_iterator(I i);
  
  // Returns: move_iterator<I>(i).
29.4.6.3.3 Class template move_sentinel

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.

Example: A move_if algorithm is easily implemented with copy_if using move_iterator and move_sentinel:

```cpp
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);
}
```

namespace std::ranges

```cpp
namespace experimental {
namespace ranges {
  class move_sentinel {
    public:
      constexpr move_sentinel();
      explicit move_sentinel(S s);
    template <ConvertibleTo<S> U>
      move_sentinel(const move_sentinel<U>& s);
    template <ConvertibleTo<S> U>
      move_sentinel& operator=(const move_sentinel<U>& s);

    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);
  }
}
}
```

29.4.6.3.4 move_sentinel operations

29.4.6.3.4.1 move_sentinel constructors

```cpp
constexpr move_sentinel();
```
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.

```cpp
explicit move_sentinel(S s);
```

Effects: Constructs a `move_sentinel`, initializing `last` with `s`.

```cpp
template <ConvertibleTo<S> U>
move_sentinel(const move_sentinel<ConvertibleTo<S>>& s);
```

Effects: Constructs a `move_sentinel`, initializing `last` with `s.last`.

### 29.4.6.3.4.2 `move_sentinel::operator=`

```cpp
template <ConvertibleTo<S> U>
move_sentinel& operator=(const move_sentinel<ConvertibleTo<S>>& s);
```

Effects: Assigns `s.last` to `last`.

Returns: `*this`.

### 29.4.6.3.4.3 `move_sentinel` comparisons

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

Effects: Equivalent to: `return i.current == s.last;`

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

Effects: Equivalent to: `return !(i == s);`

### 29.4.6.3.4.4 `move_sentinel` non-member functions

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

Effects: Equivalent to: `return s.last - i.current;`

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

Effects: Equivalent to: `return i.current - s.last;`

```cpp
template <Semiregular S>
constexpr move_sentinel<S>(S s);
```

Returns: `move_sentinel<S>(s)`.

### 29.4.6.4 Common iterators

[Editor’s note: TODO: respecify this in terms of `std::variant`.

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.

[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]

[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—

29.4.6.4.1 Class template common_iterator
namespace std::ranges {
    namespace experimental {
        namespace ranges {
            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;
                };

            };
        };
    };
}
29.4.6.4.2 common_iterator operations

29.4.6.4.2.1 common_iterator constructors

```c++
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>);
4
Effects: Constructs a common_iterator, initializing is_sentinel with u.is_sentinel, iter with u.iter, and sentinel with u.sentinel.
```

29.4.6.4.2.2 common_iterator::operator=

```c++
common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>& u>);
1
Effects: Assigns u.is_sentinel to is_sentinel, u.iter to iter, and u.sentinel to sentinel.

Returns: *this

29.4.6.4.2.3 common_iterator::operator*

```c++
decltype(auto) operator*();
decltype(auto) operator*() const
requires dereferenceable <const I>;
1
Requires: !is_sentinel
2
Effects: Equivalent to: return *iter;
```
29.4.6.4.2.4 common_iterator::operator->

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(refererence_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<reference_t<I>>::value ||
Constructible<value_type_t<I>, reference_t<I>>)

29.4.6.4.2.5 common_iterator::operator++

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;

29.4.6.4.2.6 common_iterator comparisons

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);

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);

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);

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);

29.4.6.4.2.7 iter_move

friend rvalue_reference_t<I> iter_move(const common_iterator& i)
noexcept(see below)
requires InputIterator<I>;

Requires: !i.is_sentinel.

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

Remarks: The expression in noexcept is equivalent to:
noexcept(ranges::iter_move(i.iter))

29.4.6.4.2.8 iter_swap

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

Requires: !x.is_sentinel && !y.is_sentinel.

Effects: Equivalent to ranges::iter_swap(x.iter, y.iter).

Remarks: The expression in noexcept is equivalent to:
noexcept(ranges::iter_swap(x.iter, y.iter))

29.4.6.5 Default sentinels

29.4.6.5.1 Class default_sentinel

namespace std::ranges {
  namespace experimental {
    namespace ranges {
      inline namespace v1 {
      class default_sentinel {
      }
    }
  }
}
}
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` (29.4.6.6.1)).

29.4.6.6 Counted iterators

29.4.6.6.1 Class template `counted_iterator`

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 \textit{a priori}.

[Example:

```cpp
default_sentinel
v(make_counted_iterator(s.begin(), 10), default_sentinel()); // copies 10 strings into v
```
]

Two values \( i_1 \) and \( i_2 \) of (possibly differing) types `counted_iterator<I1>` and `counted_iterator<I2>` refer to elements of the same sequence if and only if \( \text{next}(i_1\text{.base()}, i_1\text{.count()}) \) and \( \text{next}(i_2\text{.base()}, i_2\text{.count()}) \) refer to the same (possibly past-the-end) element.

```cpp
namespace std::ranges { namespace experimental { namespace ranges {
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>>& i);
    template <ConvertibleTo<I> U>
    constexpr counted_iterator& operator=(const counted_iterator<ConvertibleTo<I>>& 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 BidirectionalIterator<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 I1>
constexpr bool operator==(const counted_iterator<I1>& x, default_sentinel);

template <class I1>
constexpr bool operator==(default_sentinel, const counted_iterator<I1>& 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 I1>
constexpr bool operator!=(const counted_iterator<I1>& x, default_sentinel);

template <class I1>
constexpr bool operator!=(default_sentinel x, const counted_iterator<I1>& 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 I1>
constexpr difference_type_t<I> operator-(
```cpp
class counted_iterator

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);

}]]

29.4.6.6.2 counted_iterator operations  
29.4.6.6.2.1 counted_iterator constructors

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>>& i);

4  Effects: Constructs a counted_iterator, initializing current with i.current and cnt with i.cnt.

29.4.6.6.2.2 counted_iterator::operator=

    template <ConvertibleTo<I> U>
    constexpr counted_iterator& operator=(const counted_iterator<ConvertibleTo<I>>& i);

    1  Effects: Assigns i.current to current and i.cnt to cnt.

29.4.6.6.2.3 counted_iterator base() const;

    Returns: current.

29.4.6.6.2.4 counted Iterator count

    constexpr difference_type_t<I> count() const;

    Returns: cnt.

29.4.6.6.2.5 counted_iterator::operator*

    constexpr decltype(auto) operator*();

    constexpr decltype(auto) operator*() const
        requires dereferenceable <const I>;

    1  Effects: Equivalent to: return *current;

29.4.6.6.2.6 counted_iterator::operator++

    constexpr counted_iterator& operator++();

    Requires: cnt > 0

    Effects: Equivalent to:
        ++current;
        --cnt;
```
Returns: *this.

decltype(auto) operator++(int);

Requires: cnt > 0.
Effects: Equivalent to:

--cnt;
try {
    return current++;}
catch(...) { ++cnt; throw; }

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

Requires: cnt > 0
Effects: Equivalent to:
counted_iterator tmp = *this;
***this;
return tmp;

29.4.6.6.2.7 counted_iterator::operator--

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

Effects: Equivalent to:

--current;
++cnt;

Returns: *this.

counted_iterator operator--(int)
requires BidirectionalIterator<I>;

Effects: Equivalent to:
counted_iterator tmp = *this;
***this;
return tmp;

29.4.6.6.2.8 counted_iterator::operator+

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

Requires: n <= cnt
Effects: Equivalent to: return counted_iterator(current + n, cnt - n);

29.4.6.6.2.9 counted_iterator::operator+=

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

Requires: n <= cnt
Effects:
current += n;
cnt -= n;

Returns: *this.

29.4.6.6.2.10 counted_iterator::operator-

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

Requires: -n <= cnt
Effects: Equivalent to: return counted_iterator(current - n, cnt + n);
29.4.6.6.2.11 counted_iterator::operator-=

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

1 Requires: -n <= cnt
2 Effects:
   current -= n;
   cnt += n;
3 Returns: *this.

29.4.6.6.2.12 counted_iterator::operator[]

constexpr decltype(auto) operator[](difference_type n) const
requires RandomAccessIterator<I>;

1 Requires: n <= cnt
2 Effects: Equivalent to: return current[n];

29.4.6.6.2.13 counted_iterator comparisons

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 (29.4.6.6).
2 Effects: Equivalent to: return x.cnt == y.cnt;

template <class I>
constexpr bool operator==(const counted_iterator<I>& x, default_sentinel);

template <class I>
constexpr bool operator==(default_sentinel, const counted_iterator<I>& x);
3 Effects: Equivalent to: return x.cnt == 0;

4 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>
constexpr bool operator!=(const counted_iterator<I1>& x, default_sentinel);

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

6 template <class I1, class I2>
requires Common<I1, I2>
constexpr bool operator<(const counted_iterator<I1>& x, const counted_iterator<I2>& y);

7 Requires: x and y shall refer to elements of the same sequence (29.4.6.6).
8 Effects: Equivalent to: return y.cnt < x.cnt;

[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);

  Requires: x and y shall refer to elements of the same sequence (29.4.6.6).
  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);

  Requires: x and y shall refer to elements of the same sequence (29.4.6.6).
  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);

  Requires: x and y shall refer to elements of the same sequence (29.4.6.6).
  Effects: Equivalent to: return !(x < y);

29.4.6.6.2.14 counted_iterator non-member functions [range.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);

  Requires: x and y shall refer to elements of the same sequence (29.4.6.6).
  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);

  Effects: Equivalent to: return -x.cnt;

template <class I> 
constexpr difference_type_t<I> operator-(
    default_sentinel x, const counted_iterator<I>& y);

  Effects: Equivalent to: return y.cnt;

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

  Requires: n <= x.cnt.
  Effects: Equivalent to: return x + n;

friend constexpr rvalue_reference_t<I> iter_move(const counted_iterator& i)
    noexcept(see below)
    requires InputIterator<I>;

  Effects: Equivalent to: return ranges::iter_move(i.current);
  Remarks: The expression in noexcept is equivalent to:
    noexcept(ranges::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);
Effects: Equivalent to `ranges::iter_swap(x.current, y.current)`.

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

```cpp
noexcept(ranges::iter_swap(x.current, y.current))
```

template <Iterator I>
constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);

Requires: `n >= 0`.

Returns: `counted_iterator<I>(i, n)`.

29.4.6.7 Unreachable sentinel

29.4.6.7.1 Class unreachable

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`.

```
namespace std::ranges { namespace experimental { namespace ranges { 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; 
    } } } } } }
```

29.4.6.7.2 unreachable operations

29.4.6.7.2.1 operator==

```
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; 
```

```
1  Returns: false.
```

29.4.6.7.2.2 operator!=

```
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.
```

29.4.7 Stream iterators

```
namespace std::ranges { namespace experimental { namespace ranges { 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; 
    } } } } } }
```

```
29.4.7 Stream iterators

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]

29.4.7.1 Class template istream_iterator

The class template `istream_iterator` is an input iterator (29.4.2.11) 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.

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.

```cpp
namespace std::ranges {
    namespace experimental {
        namespace ranges {
            inline namespace v1 {
                template <class T, class charT = char, class traits = char_traits<charT>,
                class Distance = ptrdiff_t>
                class istream_iterator {
                    public:
                        using iterator_category = input_iterator_tag;
                        using difference_type = Distance;
                        using value_type = T;
                        using reference = const T&;
                        using pointer = const T*;
                        using char_type = charT;
                        using traits_type = traits;
                        using istream_type = basic_istream<charT, traits>;
                        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
                };
            }
        }
    }
}
```
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);

29.4.7.1.1 istream_iterator constructors and destructor  [range.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.

29.4.7.1.2 istream_iterator operations  [range.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);

85
### 29.4.7.2 Class template `ostream_iterator` [range.ostream.iterator]

`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

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

`ostream_iterator` is defined as:

```cpp
namespace std::ranges
{
namespace experimental
{
namespace ranges
{
inline namespace v1
{

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

public:
using difference_type = ptrdiff_t;
using char_type = charT;
using traits_type = traits;
using ostream_type = basic_ostream<charT, traits>;

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
};
}
}
}
}
```

#### 29.4.7.2.1 `ostream_iterator` constructors and destructor [range.ostream.iterator.cons.des]

```cpp
constexpr ostream_iterator() noexcept;
```

1. **Effects:** Initializes `out_stream` and `delim` with `nullptr`.

```cpp
ostream_iterator(ostream_type& s) noexcept;
```

1. **Effects:** Initializes `out_stream` with `&s` and `delim` with `nullptr`.
ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;

Effects: Initializes out_stream with &s and delim with delimiter.

ostream_iterator(const ostream_iterator& x) noexcept;

Effects: Constructs a copy of x.

~ostream_iterator();

Effects: The iterator is destroyed.

29.4.7.2.2 ostream_iterator operations

ostream_iterator& operator=(const T& value);

Effects: Equivalent to:
    *out_stream << value;
    if(delim != nullptr)
        *out_stream << delim;
    return *this;

ostream_iterator& operator*();

Returns: *this.

ostream_iterator& operator++();

ostream_iterator& operator++(int);

Returns: *this.

29.4.7.3 Class template istreambuf_iterator

The class template istreambuf_iterator defines an input iterator (29.4.2.11) 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.

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::ranges { namespace experimental { namespace ranges { inline namespace v1 { template <class charT, class traits = char_traits<charT>>
    class istreambuf_iterator {
    public:
        using iterator_category = input_iterator_tag;
        using value_type = charT;
        using difference_type = typename traits::off_type;
        using reference = charT;
        using pointer = unspecified;
        using char_type = charT;
        using traits_type = traits;
        using int_type = typename traits::int_type;
        using streambuf_type = basic_streambuf<char_type, traits>;
        using istream_type = basic_istream<charT, traits>;

        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:
    class proxy;  // exposition only
    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);

29.4.7.3.1 Class template istreambuf_iterator::proxy

namespace std::ranges {
    namespace {  // exposition only
        namespace experimental {
            namespace {  // exposition only
                namespace ranges {  // exposition only
                    namespace v1 {  // exposition only
                        template <class charT, class traits = char_traits<charT>>
                        class istreambuf_iterator<charT, traits>::proxy {
                            // exposition only
                            charT keep_;  // exposition only
                            basic_streambuf<charT, traits>* sbuf_;  // exposition only
                            proxy(charT c, basic_streambuf<charT, traits>* sbuf) {
                                keep_(c), sbuf_(sbuf) { }  // exposition only
                            }
                        public:
                            charT operator*() { return keep_; }  // exposition only
                        };  // exposition only
                    }  // namespace range
                }  // namespace experimental
            }  // namespace namespace
        }  // namespace experimental
    }  // namespace std::ranges
};  // namespace std

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.

29.4.7.3.2 istreambuf_iterator constructors

constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(default_sentinel) noexcept;

1 Effects: Constructs the end-of-stream iterator.

constexpr istreambuf_iterator(basic_istream<charT, traits>& s) noexcept;
constexpr 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.

constexpr 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.
29.4.7.3.3 istreambuf_iterator::operator*

charT operator*() const

Returns: The character obtained via the streambuf member sbuf_->sgetc().

29.4.7.3.4 istreambuf_iterator::operator++

istreambuf_iterator&

istreambuf_iterator<CharT, Traits>::operator++();

Effects: Equivalent to sbuf_->sbumpc().

Returns: *this.

proxy istreambuf_iterator<CharT, Traits>::operator++(int);

Effects: Equivalent to: return proxy(sbuf_->sbumpc(), sbuf_);

29.4.7.3.5 istreambuf_iterator::equal

bool equal(const istreambuf_iterator& b) const;

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.

29.4.7.3.6 operator==

template <class CharT, class Traits>

bool operator==(const istreambuf_iterator<CharT, Traits>& a,
                const istreambuf_iterator<CharT, Traits>& b);

Effects: Equivalent to: return a.equal(b);

template <class CharT, class Traits>

bool operator==(default_sentinel a,
                const istreambuf_iterator<CharT, Traits>& b);

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);

Effects: Equivalent to: return a.equal(istreambuf_iterator<CharT, Traits>{});

29.4.7.3.7 operator!=

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);

Effects: Equivalent to: return !(a == b);

29.4.7.4 Class template ostreambuf_iterator

namespace std {namespace ranges { namespace experimental { namespace ranges { inline namespace v1 {

class ostreambuf_iterator {

public:

  using difference_type = ptrdiff_t;
  using char_type = CharT;
  using traits_type = Traits;
  using streambuf_type = basic_streambuf<CharT, Traits>;
  using ostream_type = basic_ostream<CharT, Traits>;

namespace std::ranges { namespace experimental { namespace ranges { inline namespace v1 { template <class CharT, class Traits = CharT> class ostreambuf_iterator {
   using difference_type = ptrdiff_t;
   using char_type = CharT;
   using traits_type = Traits;
   using streambuf_type = basic_streambuf<CharT, Traits>;
   using ostream_type = basic_ostream<CharT, Traits>;

89
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
};

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.

29.4.7.4.1 ostreambuf_iterator constructors

`constexpr ostreambuf_iterator() noexcept;`

Effects: Initializes `sbuf_` with `nullptr`.

`ostreambuf_iterator(ostream_type& s) noexcept;`

Requires: `s.rdbuf()` != `nullptr`.
Effects: Initializes `sbuf_` with `s.rdbuf()`.

`ostreambuf_iterator(streambuf_type* s) noexcept;`

Requires: `s` != `nullptr`.
Effects: Initializes `sbuf_` with `s`.

29.4.7.4.2 ostreambuf_iterator operations

`ostreambuf_iterator& operator=(charT c);`

Requires: `sbuf_` != `nullptr`.
Effects: If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.
Returns: *this.

`ostreambuf_iterator& operator*();`

Returns: *this.

`ostreambuf_iterator& operator++();`

Returns: *this.

`ostreambuf_iterator& operator++(int);`

Returns: *this.

bool failed() const noexcept;

Requires: `sbuf_` != `nullptr`.
Returns: true if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

29.5 Range access

In addition to being available via inclusion of the `<experimental/ranges/range>` header, the customization point objects in 29.5 are available when `<experimental/ranges/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 compatibility with the similarly named facilities in namespace `std`. P0970 proposes a redesign that replaces the deprecated behavior with proper support for rvalue ranges.]
29.5.1 begin

The name begin denotes a customization point object (). The expression ranges::begin(E) for some subexpression E is expression-equivalent to:

(1.1) — ranges::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::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 T meets the syntactic requirements of Iterator<T>. 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 T meets the syntactic requirements of Iterator<T> with overload resolution performed in a context that includes the declaration template <class T> void begin(auto&) = delete; and does not include a declaration of ranges::begin. If Iterator is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, ranges::begin(E) is ill-formed.

2 [Note: Whenever ranges::begin(E) is a valid expression, its type satisfies Iterator. — end note]

29.5.2 end

The name end denotes a customization point object (). The expression ranges::end(E) for some subexpression E is expression-equivalent to:

(1.1) — ranges::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::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::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::begin(E))> with overload resolution performed in a context that includes the declaration template <class T> void end(auto&) = delete; and does not include a declaration of ranges::end. If Sentinel is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, ranges::end(E) is ill-formed.

2 [Note: Whenever ranges::end(E) is a valid expression, the types of ranges::end(E) and ranges::begin(E) satisfy Sentinel. — end note]

29.5.3 cbegin

The name cbegin denotes a customization point object (). The expression ranges::cbegin(E) for some subexpression E of type T is expression-equivalent to ranges::begin(static_cast<const T&>(E)).

Use of ranges::cbegin(E) with rvalue E is deprecated. [Note: This deprecated usage exists so that ranges::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::cbegin(E) is a valid expression, its type satisfies Iterator. — end note]

29.5.4 cend

The name cend denotes a customization point object (). The expression ranges::cend(E) for some subexpression E of type T is expression-equivalent to ranges::end(static_cast<const T&>(E)).

Use of ranges::cend(E) with rvalue E is deprecated. [Note: This deprecated usage exists so that ranges::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::cend(E) is a valid expression, the types of ranges::cend(E) and ranges::cbegin(E) satisfy Sentinel. — end note]
29.5.5 rbegin

The name `rbegin` denotes a customization point object (). The expression `ranges::rbegin(E)` for some subexpression `E` is expression-equivalent to:

1. `ranges::rbegin(static_cast<const T&>(E))` if `E` is an lvalue of type `T`. This usage is deprecated. [Note: This deprecated usage exists so that `ranges::rbegin(E)` behaves similarly to `std::rbegin(E)` as defined in ISO/IEC 14882 when `E` is an lvalue. —end note]

2. `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.

3. `make_reverse_iterator(ranges::end(E))` if both `ranges::begin(E)` and `ranges::end(E)` are valid expressions of the same type `I` which meets the syntactic requirements of `BidirectionalIterator<I>` (29.4.2.14).

4. Otherwise, `ranges::rbegin(E)` is ill-formed.

29.5.6 rend

The name `rend` denotes a customization point object (). The expression `ranges::rend(E)` for some subexpression `E` is expression-equivalent to:

1. `ranges::rend(static_cast<const T&>(E))` if `E` is an lvalue of type `T`. This usage is deprecated. [Note: This deprecated usage exists so that `ranges::rend(E)` behaves similarly to `std::rend(E)` as defined in ISO/IEC 14882 when `E` is an lvalue. —end note]

2. `DECAY_COPY((E).rend())` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(ranges::rbegin(E))>`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

3. `make_reverse_iterator(ranges::begin(E))` if both `ranges::begin(E)` and `ranges::end(E)` are valid expressions of the same type `I` which meets the syntactic requirements of `BidirectionalIterator<I>` (29.4.2.14).

4. Otherwise, `ranges::rend(E)` is ill-formed.

29.5.7 crbegin

The name `crbegin` denotes a customization point object (). The expression `ranges::crbegin(E)` for some subexpression `E` is expression-equivalent to `ranges::rbegin(static_cast<const T&>(E))`.

Use of `ranges::crbegin(E)` with rvalue `E` is deprecated. [Note: This deprecated usage exists so that `ranges::crbegin(E)` behaves similarly to `std::crebegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. —end note]

3. [Note: Whenever `ranges::crbegin(E)` is a valid expression, the types of `ranges::crbegin(E)` and `ranges::rbegin(E)` satisfy `Sentinel`. —end note]

29.5.8 crend

The name `crend` denotes a customization point object (). The expression `ranges::crend(E)` for some subexpression `E` of type `T` is expression-equivalent to `ranges::rend(static_cast<const T&>(E))`.

Use of `ranges::crend(E)` with rvalue `E` is deprecated. [Note: This deprecated usage exists so that `ranges::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::crend(E)` is a valid expression, the types of `ranges::crend(E)` and `ranges::crbegin(E)` satisfy `Sentinel`. —end note]

29.6 Range primitives

In addition to being available via inclusion of the `<experimental/ranges/range>` header, the customization point objects in 29.6 are available when `<experimental/ranges/iterator>` is included.
29.6.1 size

The name size denotes a customization point object (). The expression ranges::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> (29.7.3) 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::size and disable_sized_range<T> is false.

(1.4) — Otherwise, DECAY_COPY (ranges::cend(E) - ranges::cbegin(E)), except that E is only evaluated once, if it is a valid expression and the types I and S of ranges::cbegin(E) and ranges::cend(E) meet the syntactic requirements of SizedSentinel<S, I> (29.4.2.10) and ForwardIterator<I>. If SizedSentinel and ForwardIterator are not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, ranges::size(E) is ill-formed.

2 [ Note: Whenever ranges::size(E) is a valid expression, its type satisfies Integral. — end note ]

29.6.2 empty

The name empty denotes a customization point object (). The expression ranges::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::size(E) == 0 if it is a valid expression.

(1.3) — Otherwise, bool(ranges::begin(E) == ranges::end(E)), except that E is only evaluated once, if it is a valid expression and the type of ranges::begin(E) satisfies ForwardIterator.

(1.4) — Otherwise, ranges::empty(E) is ill-formed.

2 [ Note: Whenever ranges::empty(E) is a valid expression, it has type bool. — end note ]

29.6.3 data

The name data denotes a customization point object (). The expression ranges::data(E) for some subexpression E is expression-equivalent to:

(1.1) — ranges::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::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::begin(E) if it is a valid expression of pointer to object type.

(1.4) — Otherwise, ranges::data(E) is ill-formed.

2 [ Note: Whenever ranges::data(E) is a valid expression, it has pointer to object type. — end note ]

29.6.4 cdata

The name cdata denotes a customization point object (). The expression ranges::cdata(E) for some subexpression E of type T is expression-equivalent to ranges::data(static_cast<const T&>(E)).

Use of ranges::cdata(E) with rvalue E is deprecated. [ Note: This deprecated usage exists so that ranges::cdata(E) has behavior consistent with ranges::data(E) when E is an rvalue. — end note ]

3 [ Note: Whenever ranges::cdata(E) is a valid expression, it has pointer to object type. — end note ]
29.7 Range requirements

29.7.1 General

Ranges are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. Their simplest form, a range object is one on which one can call begin and end to get an iterator (29.4.2.8) and a sentinel (29.4.2.9). 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.

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 20.

Table 20 — Relations among range categories

<table>
<thead>
<tr>
<th>Sized Range</th>
<th>Range</th>
<th>View</th>
</tr>
</thead>
</table>

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.

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 (29.4.2.15). The range categories bidirectional ranges, forward ranges, input ranges, and output ranges are defined similarly.

29.7.2 Ranges

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]

```cpp
template <class T>
class Range {
    requires(Comparable<typename remove_reference_t<T> > t) {
        ranges::begin(t); // not necessarily equality-preserving (see below)
        ranges::end(t);
    }
};
```

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 (). — end note]

(2.3) — If iterator_t<T> satisfies ForwardIterator, begin(t) is equality preserving.

[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]

29.7.3 Sized ranges

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::size(t) } -> ConvertibleTo<difference_type_t<iterator_t<T>>>;
};

Given an lvalue t of type remove_reference_t<T>, SizedRange<T> is satisfied only if:

(2.1) — ranges::size(t) is $O(1)$, does not modify t, and is equal to ranges::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]

[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 (). — end note]

29.7.4 Views

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.

[Example: Examples of Views 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 (26) 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>;

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.

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 (21.9), set (26.4.6), multiset (26.4.7), unordered_set (26.5.6), or unordered_multiset (26.5.7), 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.
29.7.5 Common ranges

[Editor’s note: We’ve renamed “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. P0789 “Range Adaptors and Utilities” will be proposing a view::common adaptor that does precisely that.]

The `CommonRange` concept specifies requirements of a `Range` type for which `begin` and `end` return objects of the same type. [Note: The standard containers (26) satisfy `CommonRange`. — end note]

```cpp
template <class T>
concept bool CommonRange =
    Range<T> && Same<iterator_t<T>, sentinel_t<T>>;
```

29.7.6 Input ranges

The `InputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that satisfies `InputIterator` (29.4.2.11).

```cpp
template <class T>
concept bool InputRange =
    Range<T> && InputIterator<iterator_t<T>>;
```

29.7.7 Output ranges

The `OutputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that satisfies `OutputIterator` (29.4.2.12).

```cpp
template <class R, class T>
concept bool OutputRange =
    Range<R> && OutputIterator<iterator_t<R>, T>;
```

29.7.8 Forward ranges

The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that satisfies `ForwardIterator` (29.4.2.13).

```cpp
template <class T>
concept bool ForwardRange =
    InputRange<T> && ForwardIterator<iterator_t<T>>;
```

29.7.9 Bidirectional ranges

The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that satisfies `BidirectionalIterator` (29.4.2.14).

```cpp
template <class T>
concept bool BidirectionalRange =
    ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;
```

29.7.10 Random access ranges

The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that satisfies `RandomAccessIterator` (29.4.2.15).

```cpp
template <class T>
concept bool RandomAccessRange =
    BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```
29.8 Dangling wrapper

29.8.1 Class template dangling

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.

```cpp
namespace std::ranges { namespace experimental { namespace ranges {
    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>>;
}
}}
```

29.8.1.1 dangling operations

29.8.1.1.1 dangling constructors

- `constexpr dangling()` requires `DefaultConstructible<T>`;

  **Effects:** Constructs a dangling, value-initializing value.

- `constexpr dangling(T t);`

  **Effects:** Constructs a dangling, initializing value with `t`.

29.8.1.1.2 dangling::get_unsafe

- `constexpr T get_unsafe() const;`

  **Returns:** `value`.

29.9 Algorithms library

29.9.1 General

1 This subclause describes components that C++ programs may use to perform algorithmic operations on `containers` (Clause 26) and other sequences `ranges`.

2 The following subclauses describe components for non-modifying sequence operations, modifying sequence operations, and sorting and related operations, as summarized in Table 21.

Table 21 — Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.9.2 Non-modifying sequence operations</td>
<td><code>&lt;range&gt;</code></td>
</tr>
<tr>
<td>29.9.3 Mutating sequence operations</td>
<td><code>&lt;experimental/ranges/algorithm&gt;</code></td>
</tr>
<tr>
<td>29.9.4 Sorting and related operations</td>
<td></td>
</tr>
</tbody>
</table>

3 To ease transition, implementations provide additional algorithm signatures that are deprecated in this document (Annex ISO/IEC TS 21425-2017 §A.3).

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 The function templates defined in this subclause are not found by argument-dependent name lookup (6.4.2). When found by unqualified (6.4.1) name lookup for the `postfix-expression` in a function call (8.5.1.2), they inhibit argument-dependent name lookup.

97
Example:

```cpp
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1,2,3};
    find(begin(vec), end(vec), 2);  // #1
}
```

The function call expression at #1 invokes `std::ranges::find`, not `std::find`, despite that (a) the iterator type returned from `begin(vec)` and `end(vec)` may be associated with namespace `std` and (b) `std::find` is more specialized (17.5.6.2) than `std::ranges::find` since the former requires its first two parameters to have the same type. —end example]

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.

Both in-place and copying versions are provided for certain algorithms. 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).

[ 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>` (23.14.5), or some equivalent solution. —end note ]

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

```cpp
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of b-a is the same as of

```cpp
return distance(a, b);
```

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:

```cpp
I tmp = first;
while(tmp != last)
  ++tmp;
return tmp;
```

Overloads of algorithms that take Range arguments (29.7.2) 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.

The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

29.9.2 Non-modifying sequence operations [range.alg.nonmodifying]

29.9.2.1 All of [range.alg.all_of]

```cpp
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());
```

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`. 

98
Returns: true if \([\text{first}, \text{last})\) is empty or if \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i))\) is true for every iterator \(i\) in the range \([\text{first}, \text{last})\), and false otherwise.

Complexity: At most \(\text{last} - \text{first}\) applications of the predicate and \(\text{last} - \text{first}\) applications of the projection.

### 29.9.2.2 Any of

\[
\text{any_of}(\text{first}, \text{last}, \text{pred}, \text{proj}) = \begin{cases} \\
\text{true} & \text{if } \text{invoke}(\text{proj}, \ast i) \text{ is true for every } i \text{ in the range } [\text{first}, \text{last}), \\
\text{false} & \text{otherwise.} 
\end{cases}
\]

#### template

\[
\begin{align*}
\text{template} & \langle \text{InputIterator } I, \text{ Sentinel}\langle I\rangle S, \text{ class } \text{Proj} = \text{identity}, \\
& \text{IndirectUnaryPredicate}<\text{projected}\langle I, \text{Proj}\rangle> \text{Pred} > \\
& \text{bool} \text{any_of}(I \text{ first, } S \text{ last, } \text{Pred} \text{ pred, } \text{Proj} \text{ proj} = \text{Proj}()); \\
\end{align*}
\]

\[
\text{template} \langle \text{InputRange } Rng, \text{ class } \text{Proj} = \text{identity}, \\
\text{IndirectUnaryPredicate}<\text{projected}\langle \text{iterator}_t\langle Rng\rangle, \text{Proj}\rangle> \text{Pred} > \\
\text{bool} \text{any_of}(Rng&\& \text{ rng, } \text{Pred} \text{ pred, } \text{Proj} \text{ proj} = \text{Proj}()); \\
\]

\[
\text{Returns: false if } [\text{first}, \text{last}) \text{ is empty or if there is no iterator } i \text{ in the range } [\text{first}, \text{last}) \text{ such that } \text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)) \text{ is true, and false otherwise.}
\]

\[
\text{Complexity: At most } \text{last} - \text{first} \text{ applications of the predicate and last - first applications of the projection.}
\]

### 29.9.2.3 None of

\[
\text{none_of}(\text{first}, \text{last}, \text{pred}, \text{proj}) = \begin{cases} \\
\text{false} & \text{if } \text{invoke}(\text{proj}, \ast i) \text{ is false for every } i \text{ in the range } [\text{first}, \text{last}), \\
\text{true} & \text{otherwise.} 
\end{cases}
\]

#### template

\[
\begin{align*}
\text{template} & \langle \text{InputIterator } I, \text{ Sentinel}\langle I\rangle S, \text{ class } \text{Proj} = \text{identity}, \\
& \text{IndirectUnaryPredicate}<\text{projected}\langle I, \text{Proj}\rangle> \text{Pred} > \\
& \text{bool} \text{none_of}(I \text{ first, } S \text{ last, } \text{Pred} \text{ pred, } \text{Proj} \text{ proj} = \text{Proj}()); \\
\end{align*}
\]

\[
\text{template} \langle \text{InputRange } Rng, \text{ class } \text{Proj} = \text{identity}, \\
\text{IndirectUnaryPredicate}<\text{projected}\langle \text{iterator}_t\langle Rng\rangle, \text{Proj}\rangle> \text{Pred} > \\
\text{bool} \text{none_of}(Rng&\& \text{ rng, } \text{Pred} \text{ pred, } \text{Proj} \text{ proj} = \text{Proj}()); \\
\]

\[
\text{Returns: true if } [\text{first}, \text{last}) \text{ is empty or if } \text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)) \text{ is true for every }
\]

\[
\text{iterator } i \text{ in the range } [\text{first}, \text{last}) \text{, and false otherwise.}
\]

\[
\text{Complexity: At most last - first applications of the predicate and last - first applications of the projection.}
\]

### 29.9.2.4 For each

\[
\text{for_each}(\text{first}, \text{last}, \text{f}, \text{proj}) = \begin{cases} \\
\text{for_each}(\text{first}, \text{last} - 1, \text{f}, \text{Proj} \text{ proj} = \text{Proj}()); \\
\end{cases}
\]

\[
\text{Effects: Calls } \text{invoke}(\text{f}, \text{invoke}(\text{proj}, \ast i)) \text{ for every iterator } i \text{ in the range } [\text{first}, \text{last}), \text{starting from first and proceeding to last} - 1. \text{ [Note: If the result of } \text{invoke}(\text{proj}, \ast i) \text{ is a mutable reference, } \text{f} \text{ may apply nonconstant functions. — end note]}
\]

\[
\text{Returns: } \text{last, std::move(f)}. \\
\text{Complexity: Applies } f \text{ and proj exactly last - first times.} \\
\text{Remarks: If } f \text{ returns a result, the result is ignored.}
\]

\[
\text{[Note: The requirements of this algorithm are more strict than those specified in 28.5.4. This algorithm requires } \text{Fun} \text{ to satisfy } \text{Cpp98CopyConstructible}, \text{ whereas the algorithm in the C++ Standard requires only } \text{Cpp98MoveConstructible}. \text{ — end note]}
\]

### 29.9.2.5 Find

\[
\text{find}(\text{first}, \text{last}, \text{value}, \text{proj} \text{ proj} = \text{Proj}()); \\
\]

\[
\text{template} \langle \text{InputIterator } I, \text{ Sentinel}\langle I\rangle S, \text{ class } T, \text{ class } \text{Proj} = \text{identity} > \\
\text{requires } \text{IndirectRelation}<\text{equal_to<}, \text{projected}\langle I, \text{Proj}\rangle, \text{const } T*> \\
I \text{ find}(I \text{ first, } S \text{ last, } \text{const } T\& \text{ value, } \text{Proj} \text{ proj} = \text{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.

29.9.2.6 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<Rng1>, 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.

29.9.2.7 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>>, Proj1>,
Effects: Finds an element that matches one of a set of values.

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: \( \text{invoke(pred, invoke(proj1, *i), invoke(proj2, *j))} \neq \text{false} \). Returns last1 if [first2, last2) is empty or if no such iterator is found.

Complexity: At most \((last1-first1) \times (last2-first2)\) applications of the corresponding predicate and the two projections.

### 29.9.2.8 Adjacent find

#### [range.alg.adjacent.find]

```cpp
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{});
```

```cpp
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: \( \text{invoke(pred, invoke(proj, *i), invoke(proj, *(i + 1)))} \neq \text{false} \). Returns last if no such iterator is found.

2 Complexity: For a nonempty range, exactly \(\min((i - \text{first}) + 1, (\text{last} - \text{first}) - 1)\) applications of the corresponding predicate, where i is adjacent_find's return value, and no more than twice as many applications of the projection.

### 29.9.2.9 Count

#### [range.alg.count]

```cpp
template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>DIFFERENCE_TYPE_I
Difference_type_t<I>
count(I first, S last, const T& value, Proj proj = Proj{});
```

```cpp
template <InputRange Rng, class T, class Proj = identity>
requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>DIFFERENCE_TYPE_RNG
Difference_type_t<iterator_t<Rng>>
count(Rng&& rng, const T& value, Proj proj = Proj{});
```

```cpp
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{});
```

```cpp
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: \( \text{invoke(proj, *i)} \neq \text{value, invoke(pred, invoke(proj, *i))} \neq \text{false} \).

2 Complexity: Exactly last - first applications of the corresponding predicate and projection.

### 29.9.2.10 Mismatch

#### [range.mismatch]

```cpp
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{});

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.

Complexity: At most last1 - first1 applications of the corresponding predicate and both projections.

29.9.2.11 Equal

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, class Pred = equal_to<>>, class Proj1 = identity, class Proj2 = identity>
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 Proj1 = identity, class Proj2 = identity>
bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

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.

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.

29.9.2.12 Is permutation

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2, class Pred = equal_to<>>, class Proj1 = identity, class Proj2 = identity>
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 Proj1 = identity, class Proj2 = identity>
bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
Returns: If \( \text{last1 - first1} \neq \text{last2 - first2} \), return false. Otherwise return true if there exists a permutation of the elements in the range \([\text{first2}, \text{first2} + (\text{last1} - \text{first1}))\), beginning with \( I2 \) begin, such that \( \text{equal} \left( \text{first1}, \text{last1}, \text{begin}, \text{pred}, \text{proj1}, \text{proj2} \right) \) returns true; otherwise, return false.

Complexity: No applications of the corresponding predicate and projections if:

\[ S_{\text{Sentinel}}(S1, I1) \text{ is satisfied, and} \]
\[ S_{\text{Sentinel}}(S2, I2) \text{ is satisfied, and} \]
\[ \text{last1 - first1} \neq \text{last2} - \text{first2}. \]

Otherwise, exactly \( \text{last1 - first1} \) applications of the corresponding predicate and projections if \( \text{equal} \left( \text{first1}, \text{last1}, \text{first2}, \text{last2}, \text{pred}, \text{proj1}, \text{proj2} \right) \) would return true; otherwise, at worst \( O(N^2) \), where \( N \) has the value \( \text{last1} - \text{first1} \).

### 29.9.2.13 Search

#### Template

```cpp
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{});
```

#### Effects
Returns: The first iterator \( i \) in the range \([\text{first1}, \text{last1} - (\text{last2} - \text{first2}))\) such that for every non-negative integer \( n \) less than \( \text{last2} - \text{first2} \) the following condition holds:

\[
\text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, \ast(i + n)), \text{invoke}(\text{proj2}, \ast((\text{first2} + n))) \neq \text{false}.
\]

Returns \( \text{first1} \) if \([\text{first2}, \text{last2})\) is empty, otherwise returns \( \text{last1} \) if no such iterator is found.

Complexity: At most \( (\text{last1} - \text{first1}) \ast (\text{last2} - \text{first2}) \) applications of the corresponding predicate and projections.

---

### 29.9.2.13 Search

#### Template

```cpp
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{});
```

#### Effects
Returns: The first iterator \( i \) in the range \([\text{first1}, \text{last1} - (\text{last2} - \text{first2}))\) such that for every non-negative integer \( n \) less than \( \text{last2} - \text{first2} \) the following condition holds:

\[
\text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, \ast(i + n)), \text{invoke}(\text{proj2}, \ast((\text{first2} + n))) \neq \text{false}.
\]

Returns \( \text{first1} \) if \([\text{first2}, \text{last2})\) is empty, otherwise returns \( \text{last1} \) if no such iterator is found.

Complexity: At most \( \text{last} - \text{first} \) applications of the corresponding predicate and projection.
29.9.3 Mutating sequence operations

29.9.3.1 Copy

```cpp
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);
```

```cpp
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);
```

**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)`.

**Returns**: `{last, result + (last - first)}`.

**Requires**: `result` shall not be in the range `[first, last)`.

**Complexity**: Exactly `last - first` assignments.

```cpp
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);
```

**Effects**: For each non-negative integer `i < n`, performs `*(result + i) = *(first + i)`.

**Returns**: `{first + n, result + n}`.

**Complexity**: Exactly `n` assignments.

```cpp
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());
```

**Let** `N` be the number of iterators `i` in the range `[first, last)` for which the condition `invoke(pred, invoke(proj, *i))` holds.

**Requires**: The ranges `[first, last)` and `[result, result + N)` shall not overlap.

**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.

**Returns**: `{last, result + N}`.

**Complexity**: Exactly `last - first` applications of the corresponding predicate and projection.

**Remarks**: Stable (20.5.5.7).

```cpp
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);
```

```cpp
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);
```
Effects: Copies elements in the range `[first, last)` into the range `[result - (last-first), result)` starting from `last - 1` and proceeding to `first`. For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

Requires: `result` shall not be in the range `[first, last]`.

Returns: `{last, result - (last - first)}`.

Complexity: Exactly `last - first` assignments.

29.9.3.2 Move

```cpp
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);
```

```cpp
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);
```

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::iter_move(first + n)`.

Returns: `{last, result + (last - first)}`.

Requires: `result` shall not be in the range `[first, last)`.

Complexity: Exactly `last - first` move assignments.

```cpp
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);
```

```cpp
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);
```

Effects: Moves elements in the range `[first, last)` into the range `[result - (last-first), result)` starting from `last - 1` and proceeding to `first`. For each positive integer `n <= (last - first)`, performs `*(result - n) = ranges::iter_move(last - n)`.

Returns: `{last, result - (last - first)}`.

Requires: `result` shall not be in the range `(first, last)`.

Complexity: Exactly `last - first` assignments.

29.9.3.3 swap

```cpp
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);
```

```cpp
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);
```

Effects: For each non-negative integer `n < min(last1 - first1, last2 - first2)` performs: `ranges::iter_swap(first1 + n, first2 + n)`.

Requires: The two ranges `[first1, last1)` and `[first2, last2)` shall not overlap. `*(first1 + n)` shall be swappable with `(())*(first2 + n)`.

5) `copy_backward` should be used instead of `copy` when `last` is in the range `[result - (last - first), result)`. 6) `move_backward` should be used instead of `move` when `last` is in the range `[result - (last - first), result)`. 105
Returns: \{first1 + n, first2 + n\}, where n is min(last1 - first1, last2 - first2).

Complexity: Exactly min(last1 - first1, last2 - first2) swaps.

29.9.3.4 Transform

\begin{verbatim}
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>,
    projected<iterator_t<Rng2>, Proj1>, 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{});
\end{verbatim}

Let N be (last1 - first1) for unary transforms, or min(last1 - first1, last2 - first2) for binary transforms.

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)))).

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].

Returns: \{first1 + N, result + N\} or make_tagged_tuple(tag::in1, tag::in2, tag::out)(first1 + N, first2 + N, result + N).

Complexity: Exactly N applications of op or binary_op and the corresponding projection(s).

Remarks: result may be equal to first1 in case of unary transform, or to first1 or first2 in case of binary transform.

29.9.3.5 Replace

\begin{verbatim}
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&> &&
\end{verbatim}

7) The use of fully closed ranges is intentional.
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());

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.

Returns: last.

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());

Requires: The ranges [first,last) and [result, result + (last - first)) shall not overlap.

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

Returns: {last, result + (last - first)}.

Complexity: Exactly last - first applications of the corresponding predicate and projection.
29.9.3.6 Fill

```cpp
template <class T, OutputIterator<const T&> O, Sentinel<O> S>
O fill(O first, S last, const T& value);
```

```cpp
template <class T, OutputRange<const T&> Rng>
safe_iterator_t<Rng> fill(Rng&& rng, const T& value);
```

```cpp
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.

29.9.3.7 Generate

```cpp
template <Iterator O, Sentinel<O> S, CopyConstructible F>
requires Invocable<F&> && Writable<O, result_of_t<F&>()::invoke_result_t<F&>>
O generate(O first, S last, F gen);
```

```cpp
template <class Rng, CopyConstructible F>
requires Invocable<F&> && OutputRange<Rng, result_of_t<F&>()::invoke_result_t<F&>>
safe_iterator_t<Rng> generate(Rng&& rng, F gen);
```

```cpp
template <Iterator O, CopyConstructible F>
requires Invocable<F&> && Writable<O, result_of_t<F&>()::invoke_result_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.

29.9.3.8 Remove

```cpp
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{});
```

```cpp
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{});
```

```cpp
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{});
```

```cpp
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{});
```
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.

Returns: The end of the resulting range.

Remarks: Stable (20.5.5.7).

Complexity: Exactly last - first applications of the corresponding predicate and projection.

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, Pred pred, Proj proj = Proj{});

Requires: The ranges [first, last) and [result, result + (last - first)) shall not overlap.

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.

Returns: A pair consisting of last and the end of the resulting range.

Complexity: Exactly last - first applications of the corresponding predicate and projection.

Remarks: Stable (20.5.5.7).

29.9.3.9 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{});

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.

Returns: The end of the resulting range.
3  Complexity: For nonempty ranges, exactly \((\text{last} - \text{first}) - 1\) applications of the corresponding predicate and no more than twice as many applications of the projection.

3  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>>) ||  
        IndirectCopyStorable<I, O>  
      )  
    tagged_pair<tag::in(I), tag::out(O)>  
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj());

3  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>>) ||  
        IndirectCopyStorable<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 \([\text{first}, \text{last})\) and \([\text{result}, \text{result}+(\text{last}-\text{first}))\) 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 \([\text{first}, \text{last})\) for which the following corresponding conditions hold:

5  \[
\text{invoke}(\text{proj}, *i) == \text{invoke}(\text{proj}, *(i - 1))
\]

5  or

5  \[
\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *i), \text{invoke}(\text{proj}, *(i - 1))) \neq \text{false}.
\]

6  Returns: A pair consisting of \text{last} and the end of the resulting range.

7  Complexity: For nonempty ranges, exactly \(\text{last} - \text{first} - 1\) applications of the corresponding predicate and no more than twice as many applications of the projection.

29.9.3.10 Reverse [range.alg.reverse]

3  template <BidirectionalIterator I, Sentinel<I> S>  
    requires Permutable<I>  
    I reverse(I first, S last);

3  template <BidirectionalRange Rng>  
    requires Permutable<iterator_t<Rng>>  
    safe_iterator_t<Rng>  
    reverse(Rng&& rng);

1  Effects: For each non-negative integer \(i < (\text{last} - \text{first})/2\), applies \text{iter_swap} to all pairs of iterators \(\text{first} + i, (\text{last} - i) - 1\).

2  Returns: last.

3  Complexity: Exactly \((\text{last} - \text{first})/2\) swaps.

3  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);

3  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 \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result}+(\text{last}-\text{first}))\) such that for every non-negative integer \(i < (\text{last} - \text{first})\) the following assignment takes place: \(*(\text{result} + (\text{last} - \text{first}) - 1 - i) = *(\text{first} + i)\).

5  Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result}+(\text{last}-\text{first}))\) shall not overlap.
29.9.3.11 Rotate

Returns: \( \{ \text{last, result + (last - first)} \} \).

Complexity: Exactly \( \text{last - first} \) assignments.

```cpp
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);

Effects: For each non-negative integer \( i < (\text{last - first}) \), places the element from the position
  \( \text{first} + i \) into position \( \text{first} + (i + (\text{last - middle})) \% (\text{last - first}) \).

Returns: \( \{ \text{first + (last - middle), last} \} \).

Remarks: This is a left rotate.

Requires: \( [\text{first}, \text{middle}) \) and \( [\text{middle}, \text{last}) \) shall be valid ranges.

Complexity: At most \( \text{last - first} \) swaps.

```cpp
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);

Effects: Copies the range \( [\text{first}, \text{last}) \) to the range \( [\text{result, result + (last - first}) \) such that
  for each non-negative integer \( i < (\text{last - first}) \) the following assignment takes place:
  \( *(\text{result} + i) = *(\text{first} + (i + (\text{middle} - \text{first})) \% (\text{last - first})) \).

Returns: \( \{ \text{last, result + (last - first)} \} \).

Requires: The ranges \( [\text{first, last}) \) and \( [\text{result, result + (last - first}) \) shall not overlap.

Complexity: Exactly \( \text{last - first} \) assignments.

29.9.3.12 Shuffle

```cpp
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
requires Permutable<I> \&\&
  UniformRandomNumberBitGenerator<remove_reference_t<Gen>> \&\&
 ConvertibleTo<result_of_t<Gen&()>, invoke_result_t<Gen&>, difference_type_t<I>>
I shuffle(I first, S last, Gen& g);

template <RandomAccessRange Rng, class Gen>
requires Permutable<iterator_t<Rng>> \&\&
  UniformRandomNumberBitGenerator<remove_reference_t<Gen>> \&\&
 ConvertibleTo<result_of_t<Gen&()>, invoke_result_t<Gen&>, difference_type_t<I>>
safe_iterator_t<Rng>
  shuffle(Rng&& rng, Gen& g);

Effects: Permutates the elements in the range \( [\text{first, last}) \) such that each possible permutation of
  those elements has equal probability of appearance.

Complexity: Exactly \( (\text{last} - \text{first}) - 1 \) swaps.

Returns: last

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.
```
29.9.3.13 Partitions

```cpp
#include <algorithm>

// Template for checking partitioning
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 for checking partitioning on ranges
template <InputRange Rng, class Proj = identity,
          IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
bool is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});

// Template for partitioning
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 for partitioning on ranges
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 for stable partitioning
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 for stable partitioning on ranges
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 for partitioning using copy
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{});
```

---

1. **Returns:** true if \([\text{first}, \text{last})\) is empty or if \([\text{first}, \text{last})\) is partitioned by \(\text{pred}\) and \(\text{proj}\), i.e. if all iterators \(i\) for which \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *i)) != \text{false}\) come before those that do not, for every \(i\) in \([\text{first}, \text{last})\).

2. **Complexity:** Linear. At most last − first applications of \(\text{pred}\) and \(\text{proj}\).

3. **Effects:** Permutes the elements in the range \([\text{first}, \text{last})\) such that there exists an iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *j)) != \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *k)) == \text{false}\).

4. **Returns:** An iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *j)) != \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *k)) == \text{false}\).

5. **Complexity:** If \(\text{I}\) meets the requirements for a BidirectionalIterator, at most \((\text{last} - \text{first}) / 2\) swaps; otherwise at most last − first swaps. Exactly last − first applications of the predicate and projection.

6. **Effects:** Permutes the elements in the range \([\text{first}, \text{last})\) such that there exists an iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *j)) != \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *k)) == \text{false}\).

7. **Returns:** An iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *j)) != \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *k)) == \text{false}\). The relative order of the elements in both groups is preserved.

8. **Complexity:** At most \((\text{last} − \text{first}) \ast \log(\text{last} − \text{first})\) swaps, but only linear number of swaps if there is enough extra memory. Exactly last − first applications of the predicate and projection.
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{});

\textbf{Requires:} The input range shall not overlap with either of the output ranges.

\textbf{Effects:} For each iterator \( i \) in \( [\text{first}, \text{last}) \), copies \( *i \) to the output range beginning with \text{out\_true} if \( \text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)) \) is true, or to the output range beginning with \text{out\_false} otherwise.

\textbf{Returns:} A tuple \( p \) such that \( \text{get\:<0\>(p)} \) is \text{last}, \( \text{get\:<1\>(p)} \) is the end of the output range beginning at \text{out\_true}, and \( \text{get\:<2\>(p)} \) is the end of the output range beginning at \text{out\_false}.

\textbf{Complexity:} Exactly \( \text{last} - \text{first} \) applications of \text{pred} and \text{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{});

\textbf{Requires:} \([\text{first}, \text{last})\] shall be partitioned by \text{pred} and \text{proj}, i.e. there shall be an iterator \text{mid} such that \( \text{all\_of}(\text{first}, \text{mid}, \text{pred}, \text{proj}) \) and \( \text{none\_of}(\text{mid}, \text{last}, \text{pred}, \text{proj}) \) are both true.

\textbf{Returns:} An iterator \text{mid} such that \( \text{all\_of}(\text{first}, \text{mid}, \text{pred}, \text{proj}) \) and \( \text{none\_of}(\text{mid}, \text{last}, \text{pred}, \text{proj}) \) are both true.

\textbf{Complexity:} \( O(\log(\text{last} - \text{first})) \) applications of \text{pred} and \text{proj}.

\subsection*{29.9.4 Sorting and related operations \texttt{[range.alg.sorting]}}

All the operations in \texttt{29.9.4} take an optional binary callable predicate of type \texttt{Comp} that defaults to \texttt{less\<>}.

\textbf{Comp} is a callable object (23.14.3). The return value of the \texttt{invoke} operation applied to an object of type \texttt{Comp}, when contextually converted to \texttt{bool} (Clause 7), yields \texttt{true} if the first argument of the call is less than the second, and \texttt{false} otherwise. \texttt{Comp comp} is used throughout for algorithms assuming an ordering relation. It is assumed that \texttt{comp} will not apply any non-constant function through the dereferenced iterator.

A sequence is \textit{sorted with respect to a comparator and projection} \texttt{comp} and \texttt{proj} if for every iterator \texttt{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, \( \text{invoke}(\text{comp}, \text{invoke}(\text{proj}, \ast(i + n)), \text{invoke}(\text{proj}, \ast i)) == \text{false} \).

A sequence \([\text{start}, \text{finish})\) is \textit{partitioned with respect to an expression} \( f(e) \) if there exists an integer \( n \) such that for all \( 0 \leq i < \text{distance}(\text{start}, \text{finish}), f(*(\text{start} + i)) \) is true if and only if \( i < n \).

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 \texttt{operator==}, but an equivalence relation induced by the strict weak ordering. That is, two elements \texttt{a} and \texttt{b} are considered equivalent if and only if \(! (\text{a < b}) && !(\text{b < a}) \).

\subsection*{29.9.4.1 Sorting \texttt{[range.alg.sort]}}

\textbf{sort} \texttt{[range.sort]}

\texttt{template <RandomAccessIterator I, Sentinel<I> S, class Comp = \texttt{less\<>},
    class Proj = identity>
requires Sortable<I, Comp, Proj>
I sort(I first, S last, Comp comp = \texttt{Comp\{\}}, Proj proj = \texttt{Proj\{\}});
sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

**Effects:** Sorts the elements in the range \([\text{first, last})\).

**Returns:** last.

**Complexity:** \(O(N \log(N))\) (where \(N = \text{last} - \text{first}\)) comparisons, and twice as many applications of the projection.

### 29.9.4.1.2 stable_sort

```cpp
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
class Proj = identity>
requires Sortable<I, Comp, Proj>
I stable_sort(I first, I last, Comp comp = Comp{}, Proj proj = Proj{});
```

```cpp
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{});
```

**Effects:** Sorts the elements in the range \([\text{first, last})\).

**Returns:** last.

**Complexity:** Let \(N = \text{last} - \text{first}\). 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.

**Remarks:** Stable (20.5.5.7).

### 29.9.4.1.3 partial_sort

```cpp
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{});
```

```cpp
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{});
```

**Effects:** Places the first \(\text{middle} - \text{first}\) sorted elements from the range \([\text{first, last})\) into the range \([\text{first, middle})\). The rest of the elements in the range \([\text{middle, last})\) are placed in an unspecified order.

**Returns:** last.

**Complexity:** It takes approximately \((\text{last} - \text{first}) \times \log(\text{middle} - \text{first})\) comparisons, and exactly twice as many applications of the projection.

### 29.9.4.1.4 partial_sort_copy

```cpp
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{});
```

```cpp
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>>
```

---

114
safe_iterator_t<Rng2>

Partial sort copy (Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

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)).

Returns: The smaller of: result_last or result_first + (last - first).

Complexity: Approximately
(last - first) + log(min(last - first, result_last - result_first))

comparisons, and exactly twice as many applications of the projection.

29.9.4.1.5 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{});

Returns: is_sorted_until(first, last, comp, proj) == last

Template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
IndirectStrictWeakOrder<projected_iterator_t<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{});

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.

Complexity: Linear.

29.9.4.2 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{});

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.

Returns: last.

Complexity: Linear on average.

29.9.4.3 Binary search

Template <RandomAccessIterator I, Sentinel<I> S, class Comp = less>,
class Proj = identity>
requires Sortable<I, Comp, Proj>
I binary_search(I first, I last, S sentinel, Comp comp = Comp{}, Proj proj = Proj{});

Returns: first == last.

Complexity: Logarithmic.

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.
29.9.4.3.1 \textit{lower\_bound} \hfill [range.lower.bound]

\texttt{template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less>>}
\begin{itemize}
\item[I] \texttt{lower\_bound(I first, S last, const T& value, Comp comp = Comp{},
Proj proj = Proj{});} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{safe\_iterator\_t<Rng> lower\_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{requires: The elements \(e\) of \([first, last)\) shall be partitioned with respect to the expression \textit{invoke}\(\)
\textit{comp, invoke\(\)(proj, \(e\)), value}. \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{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: \textit{invoke}\(\)
\textit{comp, invoke\(\)(proj, \(*j\)), value} \(!=\) false. \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{complexity: At most \(\log_2(last - first) + O(1)\) applications of the comparison function and projection. \hfill (1)
\end{itemize}

29.9.4.3.2 \textit{upper\_bound} \hfill [range.upper.bound]

\texttt{template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less>>}
\begin{itemize}
\item[I] \texttt{upper\_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{safe\_iterator\_t<Rng> upper\_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{requires: The elements \(e\) of \([first, last)\) shall be partitioned with respect to the expression \textit{!invoke}\(\)
\textit{comp, value, invoke\(\)(proj, \(e\))}. \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{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: \textit{invoke}\(\)
\textit{comp, value, \textit{invoke\(\)(proj, \(*j\))} \(!=\) false. \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{complexity: At most \(\log_2(last - first) + O(1)\) applications of the comparison function and projection. \hfill (1)
\end{itemize}

29.9.4.3.3 \textit{equal\_range} \hfill [range.equal.range]

\texttt{template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less>>}
\begin{itemize}
\item[I] \texttt{tagged\_pair<tag::begin(I), tag::end(I)> equal\_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{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{});} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{requires: The elements \(e\) of \([first, last)\) shall be partitioned with respect to the expressions \textit{invoke}\(\)
\textit{comp, invoke\(\)(proj, \(e\)), value} and \textit{!invoke}\(\)
\textit{comp, value, invoke\(\)(proj, \(e\))}. Also, for all elements \(e\) of \([first, last)\), \textit{invoke}\(\)
\textit{comp, invoke\(\)(proj, \(e\)), value} shall imply \textit{!invoke}\(\)
\textit{comp, value, \textit{invoke\(\)(proj, \(e\))}. \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{returns: \{lower\_bound(first, last, value, comp, proj),
upper\_bound(first, last, value, comp, proj)} \hfill (1)
\end{itemize}
\begin{itemize}
\item[I] \texttt{complexity: At most \(2 \times \log_2(last - first) + O(1)\) applications of the comparison function and projection. \hfill (1)
\end{itemize}
29.9.4.3.4 binary_search

```cpp
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{});
```

```cpp
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 \(\log_2(last - first) + O(1)\) applications of the comparison function and projection.

29.9.4.4 Merge

```cpp
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{});
```

```cpp
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 (20.5.5.7).
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{});

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.

Requires: The ranges [first,middle) and [middle,last) shall be sorted with respect to comp and proj.

Returns: last

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.

Remarks: Stable (20.5.5.7).

29.9.4.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They also work with multisets (26.4.7) 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.

29.9.4.5.1 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{});

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.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of the comparison function and projections.

29.9.4.5.2 set_union

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
          class Proj1 = identity, class Proj2 = identity,
          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{});
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.

Requires: The resulting range shall not overlap with either of the original ranges.

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.

Complexity: At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) applications of the comparison function and projections.

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.

29.9.4.5.3 `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{});
```

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.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: The end of the constructed range.

Complexity: At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) applications of the comparison function and projections.

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.

29.9.4.5.4 `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{});
```

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.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: `{last1, result + n}`, where `n` is the number of elements in the constructed range.

Complexity: At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) applications of the comparison function and projections.
Remarks: If \([first_1, last_1)\) contains \(m\) elements that are equivalent to each other and \([first_2, last_2)\) contains \(n\) elements that are equivalent to them, the last \(\max(m - n, 0)\) elements from \([first_1, last_1)\) shall be copied to the output range.

### 29.9.4.5.5 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{});
```

Effects: Copies the elements of the range \([first_1, last_1)\) that are not present in the range \([first_2, last_2)\), and the elements of the range \([first_2, last_2)\) that are not present in the range \([first_1, last_1)\) to the range beginning at \(result\). The elements in the constructed range are sorted.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: \(\text{make_tagged_tuple}\langle\text{tag::in1}, \text{tag::in2}, \text{tag::out}\rangle(last_1, last_2, result + n)\), where \(n\) is the number of elements in the constructed range.

Complexity: At most \(2 \times ((last_1 - first_1) + (last_2 - first_2)) - 1\) applications of the comparison function and projections.

Remarks: If \([first_1, last_1)\) contains \(m\) elements that are equivalent to each other and \([first_2, last_2)\) 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 \([first_1, last_1)\) if \(m > n\), and the last \(n - m\) of these elements from \([first_2, last_2)\) if \(m < n\).

### 29.9.4.6 Heap operations

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 \(\text{pop_heap()}\), or a new element added by \(\text{push_heap()}\), in \(O(\log(N))\) time.

These properties make heaps useful as priority queues.

**make_heap()** converts a range into a heap and **sort_heap()** turns a heap into a sorted sequence.

### 29.9.4.6.1 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{});
```

Effects: Places the value in the location last - 1 into the resulting heap \([first, last)\).

Requires: The range \([first, last - 1)\) shall be a valid heap.

Returns: last

Complexity: At most \(\log(last - first)\) applications of the comparison function and projection.
29.9.4.6.2    pop_heap
              [range.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{});

            Requires: The range [first,last) shall be a valid non-empty heap.

            Effects: Swaps the value in the location first with the value in the location last - 1 and makes
                     [first,last - 1) into a heap.

            Returns: last

            Complexity: At most 2 * log(last - first) applications of the comparison function and projection.

29.9.4.6.3    make_heap
              [range.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{});

            Effects: Constructs a heap out of the range [first,last).

            Returns: last

            Complexity: At most 3 * (last - first) applications of the comparison function and projection.

29.9.4.6.4    sort_heap
              [range.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{});

            Effects: Sorts elements in the heap [first,last).

            Requires: The range [first,last) shall be a valid heap.

            Returns: last

            Complexity: At most \(N \log(N)\) comparisons (where \(N == last - first\)), and exactly twice as many
                        applications of the projection.

29.9.4.6.5    is_heap
              [range.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{});

            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{});

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.

Complexity: Linear.

29.9.4.7 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{});

Returns: The smaller value.

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{});

Requires: distance(rng) > 0.

Returns: The smallest value in the initializer_list or range.

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{});

Returns: The larger value.

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{});

Requires: distance(rng) > 0.

Returns: The largest value in the initializer_list or range.

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{});
Returns: \{b, a\} if b is smaller than a, and \{a, b\} otherwise.

Remarks: Returns \{a, b\} when the arguments are equivalent.

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<min(T), 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<min(value_type_t<iterator_t<Rng>>), max(value_type_t<iterator_t<Rng>>)> minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Requires: distance(rng) > 0.

Returns: \{x, y\}, where x has the smallest and y has the largest value in the initializer_list or range.

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.

Complexity: At most \((3/2) \times \text{distance}(rng)\) 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{});

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.

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{});

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.

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<min(I), 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<min(safe_iterator_t<Rng>),
tag: max(safe_iterator_t<Rng>)>

minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

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.

Complexity: At most \(\max(\frac{3}{2}(N - 1)), 0\) applications of the comparison function and at most twice as many applications of the projection, where \(N\) is \(distance(first, last)\).

29.9.4.8 Lexicographical comparison

\[\text{range.alg.lex.comparison}\]

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{});

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.

Complexity: At most \(2*\min((last1 - first1), (last2 - first2))\) applications of the corresponding comparison and projections.

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;

Remarks: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

29.9.4.9 Permutation generators

\[\text{range.alg.permutation.generators}\]

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{});

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.

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{});

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 \(\text{comp}\) and \(\text{proj}\).

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.

Complexity: At most \((last - first)/2\) swaps.

Annex A  (informative)

Acknowledgements

This work was made possible in part by a grant from the Standard C++ Foundation, and by the support
of the authors’ employers Facebook and Microsoft.

Bibliography

Index

constant iterator, 41
multi-pass guarantee, 48
mutable iterator, 41
projection, 4
requirements
   iterator, 40
swappable, 6
swappable with, 6
undefined behavior, 87
unspecified, 114
Index of library names

adjacent_find, 101
advance, 56
all_of, 98
any_of, 99

back_insert_iterator, 63
  back_insert_iterator, 63
  constructor, 63
back_inserter, 64
base
  counted_iterator, 79
  move_iterator, 68
  reverse_iterator, 60
bidirectional_iterator_tag, 55
BidirectionalIterator, 48
binary_search, 117
common_iterator, 73
  common_iterator, 74
  constructor, 74
  iter_move, 76
  iter_swap, 76
  operator!=, 76
  operator*, 74
  operator++, 75
  operator-, 76
  operator->, 75
  operator=, 74
  operator==, 75, 76

  copy, 104
copy_backward, 104
copy_if, 104
copy_n, 104
count, 101
  counted_iterator, 79
count_if, 101
counted_iterator, 77
  base, 79
  constructor, 79
  count, 79
  counted_iterator, 79
  iter_move, 82
  iter_swap, 82
  operator!=, 81
  operator*, 79
  operator+, 80, 82
  operator++, 79, 80
  operator=, 80
  operator-=, 80
  operator-=, 81
  operator->, 80
  operator<, 81
  operator<=, 81
  operator==, 79
operator==, 81
operator>, 82
operator>==, 82
operator[], 81
dangling, 97
dangling, 97
get_unsafe, 97
default_sentinel, 76
difference_type, 43
difference_type_t, 43
distance, 56
empty, 55
equal, 102
equal_range, 116
equal_to, 14
equal_to<>), 14
failed
  ostreambuf_iterator, 90
fill, 108
fill_n, 108
find, 99
find_end, 100
find_first_of, 100
find_if, 99
find_if_not, 99
for_each, 99
forward_iterator_tag, 55
ForwardIterator, 48
front_insert_iterator, 64
  constructor, 64
  front_insert_iterator, 64
  front_inserter, 65
generate, 108
generate_n, 108
get_unsafe
dangling, 97
greater, 14
greater<>), 15
greater_equal, 14
greater_equal<>), 15
includes, 118
Incrementable, 46
indirect_result, 50
IndirectlyComparable, 52
IndirectlyCopyable, 51
IndirectlyCopyableStorable, 52
IndirectlyMovable, 51
IndirectlyMovableStorable, 51
IndirectlySwappable, 52
IndirectRegularUnaryInvocable, 50
IndirectRelation, 50
IndirectStrictWeakOrder, 50
IndirectUnaryInvocable, 50
IndirectUnaryPredicate, 50
inplace_merge, 117
InputIterator, 47
insert_iterator, 65
   constructor, 65
   insert_iterator, 65
inserter, 66
is_heap, 121
is_heap_until, 122
is_partitioned, 112
is_permutation, 102
is_sorted, 115
is_sorted_until, 115
istream_iterator, 84
   constructor, 85
destructor, 85
operator! =, 86
operator*, 85
operator++, 85
operator->, 85
operator==, 85
istreambuf_iterator, 87
   constructor, 88
operator++, 89
iter_move
   common_iterator, 76
counted_iterator, 82
move_iterator, 70
reversed_iterator, 62
iter_swap
   common_iterator, 76
counted_iterator, 82
move_iterator, 70
reversed_iterator, 62
Iterator, 46
iterator_category, 44
iterator_category_t, 44
iterator_traits, 53
less, 14
less<>, 15
less_equal, 14
less_equal<>, 16
lexicographical_compare, 124
lower_bound, 116
make_counted_iterator, 83
make_heap, 121
make_move_iterator, 70
make_move_iterator, 72
make_reverse_iterator, 62
make_tagged_pair, 12
make_tagged_tuple, 13
max, 122
max_element, 123
merge, 117
Mergeable, 52
min, 122
min_element, 123
minmax, 122, 123
minmax_element, 123
mismatch, 101
move, 105
move_backward, 105
move_iterator, 66
   base, 68
destructor, 68
iter_move, 70
iter_swap, 70
move_iterator, 68
operator! =, 69
operator*, 68
operator++, 69, 70
operator++, 68
operator*, 69
operator*, 69, 70
operator-, 69
operator-, 69
operator<, 69
operator<, 70
operator=, 68
operator==, 69
operator>, 70
operator>=, 70
operator[] =, 69
move_sentinel, 71
   constructor, 72
move_sentinel, 71
operator! =, 72
operator-, 72
operator*, 72
operator*=, 72
next, 57
next_permutation, 124
none_of, 99
not_equal_to, 14
not_equal_to<>, 15
nth_element, 115
operator!=
   common_iterator, 76
counted_iterator, 81
istream_iterator, 86
istreambuf_iterator, 89
move_iterator, 69
move_sentinel, 72
reverse_iterator, 61
unreachable, 83
operator*
   back_insert_iterator, 64
counted_iterator, 74
counted_iterator, 79
front_insert_iterator, 65
insert_iterator, 66
istream_iterator, 85
istreambuf_iterator, 89
move_iterator, 68
ostream_iterator, 87
ostreambuf_iterator, 90
reverse_iterator, 60
operator+
  counted_iterator, 80, 82
  move_iterator, 69, 70
  reverse_iterator, 60, 62
operator++
  back_insert_iterator, 64
  common_iterator, 75
  counted_iterator, 79, 80
  front_insert_iterator, 65
  insert_iterator, 66
  istream_iterator, 85
  istreambuf_iterator, 89
  move_iterator, 68
  ostream_iterator, 87
  ostreambuf_iterator, 90
  reverse_iterator, 60
operator++=
  counted_iterator, 80
  move_iterator, 69
  reverse_iterator, 60
operator-
  common_iterator, 76
  counted_iterator, 80, 82
  move_iterator, 69, 70
  move_sentinel, 72
  reverse_iterator, 60, 62
operator-=
  counted_iterator, 81
  move_iterator, 69
  reverse_iterator, 61
operator>
  common_iterator, 75
  istream_iterator, 85
  reverse_iterator, 61
operator>=
  counted_iterator, 82
  move_iterator, 70
  reverse_iterator, 61
operator[]
  counted_iterator, 81
  move_iterator, 69
  reverse_iterator, 61
ostream_iterator, 86
  constructor, 86, 87
  destructor, 87
  operator*, 87
  operator++, 87
  operator=, 87
ostreambuf_iterator, 89
  constructor, 90
output_iterator_tag, 55
OutputIterator, 48

partial_sort, 114
partial_sort_copy, 114
partition, 112
partition_copy, 112
partition_point, 113
Permutable, 52
pop_heap, 121
prev, 57
prev_permutation, 125
projected, 51
proxy
  istreambuf_iterator, 88
push_heap, 120
random_access_iterator_tag, 55
RandomAccessIterator, 49
<range>, 16
ranges::swap, 8
Readable, 45
remove, 108
remove_copy, 109
remove_copy_if, 109
remove_if, 108
replace, 106
replace_copy, 107
replace_copy_if, 107
replace_if, 106
reverse, 110
reverse_copy, 110
reverse_iterator, 58
   reverse_iterator, 59
   base, 60
   constructor, 59
   iter_move, 62
   iter_swap, 62
   make_reverse_iterator non-member function, 62
   operator++, 60
   operator--., 60
rotate, 111
rotate_copy, 111
rvalue_reference_t, 45
search, 103
search_n, 103
Sentinel, 47
set_difference, 119
set_intersection, 119
set_symmetric_difference, 120
set_union, 118
shuffle, 111
SizedSentinel, 47
sort, 113
sort_heap, 121
Sortable, 53
stable_partition, 112
stable_sort, 114
swap
   tagged, 11
   tagged, 11
swap_ranges, 105
Swappable, 5
SwappableWith, 5
tagged, 9
   operator=, 10, 11
   swap, 11
   tagged, 10
tagged_tuple
   make_tagged_tuple, 13
transform, 106
tuple_element, 11
tuple_size, 11
unique, 109
unique_copy, 110
unreachable, 83
   operator!=, 83
operator==, 83
upper_bound, 116
value_type, 43
WeaklyIncrementable, 46
Writable, 45