Working Draft, C++ Extensions for Ranges

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

Contents  
List of Tables  
List of Figures  
1 General  
   1.1 Scope  
   1.2 References  
   1.3 Implementation compliance  
   1.4 Namespaces, headers, and modifications to standard classes  
6 Statements  
   6.5 Iteration statements  
17 Library introduction  
19 Concepts library  
   19.1 General  
   19.2 Core language concepts  
   19.3 Comparison concepts  
   19.4 Object concepts  
   19.5 Function concepts  
20 General utilities library  
   20.2 Utility components  
   20.9 Function objects  
   20.15 Tagged tuple-like types  
24 Iterators library  
   24.1 General  
   24.2 Iterator requirements  
   24.3 Indirect callable requirements  
   24.4 Common algorithm requirements  
   24.5 Iterator range requirements  
   24.6 Header `<experimental/ranges/iterator>` synopsis  
   24.7 Iterator primitives  
   24.8 Iterator adaptors  
   24.9 Stream iterators  
   24.10 Range concepts  
   24.11 Range access  
   24.12 Range primitives  
25 Algorithms library  
   25.1 General  
   25.?? Tag specifiers

Contents ii
List of Tables iv
List of Figures v
25.2 Non-modifying sequence operations ........................................ 139
25.3 Mutating sequence operations ............................................. 149
25.4 Sorting and related operations ............................................. 165
25.5 C library algorithms ......................................................... 187

26 Numerics library ............................................................... 189
  26.5 Random number generation ............................................. 189

D Compatibility features ....................................................... 190
  D.10 Range-and-a-half algorithms ......................................... 190

A Acknowledgements ............................................................ 192

B Compatibility ................................................................. 193
  B.1 C++ and Ranges ......................................................... 193
  B.2 Ranges and the Palo Alto TR (N3351) ................................ 194

C Future Work ................................................................. 196
  C.1 Proxy Iterators .......................................................... 196
  C.2 Iterator Range Type .................................................... 196
  C.3 Range Views and Actions .............................................. 196
  C.4 Range Facade and Adaptor Utilities ................................ 196
  C.5 Infinite Ranges ......................................................... 197
  C.6 Common Type .......................................................... 197
  C.7 Numeric Algorithms and Containers ................................ 197
  C.8 Verbosity in Algorithm Constraints ................................ 197
  C.9 Initializer Lists ....................................................... 198

D Reference implementation for tagged .................................. 199

Bibliography ................................................................. 202

Index ................................................................. 203

Index of library names .................................................. 204
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ranges TS library headers</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Fundamental concepts library summary</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Iterators library summary</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>Relations among iterator categories</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>Ranges library summary</td>
<td>105</td>
</tr>
<tr>
<td>6</td>
<td>Relations among range categories</td>
<td>105</td>
</tr>
<tr>
<td>7</td>
<td>Algorithms library summary</td>
<td>116</td>
</tr>
<tr>
<td>8</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
<td>187</td>
</tr>
</tbody>
</table>
List of Figures
1 General

Naturally the villagers were intrigued and soon a fire was put to the town’s greatest kettle as the soldiers dropped in three smooth stones.

“Now this will be a fine soup”, said the second soldier; “but a pinch of salt and some parsley would make it wonderful!”

—Author Unknown

1.1 Scope

This Technical Specification describes extensions to the C++ Programming Language 1.2 that permit operations on ranges of data. These extensions include changes and additions to the existing library facilities as well as the extension of some core language facilities. In particular, changes and extensions to the Standard Library include:

- The reformulation of the foundational and iterator concept requirements using the syntax of the Concepts TS 1.2.
- The respecification of the Standard Library algorithms in terms of the new concepts.
- The loosening of the algorithm constraints to permit the use of sentinels to denote the end of a range and corresponding changes to algorithm return types where necessary.
- The addition of new concepts describing range and view abstractions; that is, objects with a begin iterator and an end sentinel.
- The addition of new overloads of the Standard Library algorithms that take iterable objects.
- Support of callable objects (as opposed to function objects) passed as arguments to the algorithms.
- The addition of optional projection arguments to the algorithms to permit on-the-fly data transformations.
- Changes to existing iterator primitives and new primitives in support of the addition of sentinels to the library.
- Changes to the existing iterator adaptors and stream iterators to make them model the new iterator concepts.
- New iterator adaptors (counted_iterator and common_iterator) and sentinels (unreachable).

Changes to the core language include:

- the extension of the range-based for statement to support the new iterator range requirements (24.11).

The scope of this paper does not yet extend to the other parts of the Standard Library that need to change because of the addition of concepts to the language (e.g., the numeric algorithms), nor does it add range support to all the places that could benefit from it (e.g., the containers).

This paper does not specify any new range views, actions, or facade or adaptor utilities. See the Future Work appendix (C).
1.2 References

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

1.2.1 ISO/IEC 14882:2014, Programming Languages - C++
1.2.2 JTC1/SC22/WG21 N4377, Technical Specification - C++ Extensions for Concepts
1.2.3 JTC1/SC22/WG21 N4128, Ranges for the Standard Library, Revision 1
1.2.4 JTC1/SC22/WG21 N3351, A Concept Design for the STL

ISO/IEC 14882:2014 is herein called the C++ Standard, N3351 is called the “The Palo Alto” report, and N4377 is called the Concepts TS.

1.3 Implementation compliance

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

1.4 Namespaces, headers, and modifications to standard classes

Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they should not be declared directly within namespace std. Unless otherwise specified, all components described in this document either:

1.4.1 modify an existing interface in the C++ Standard Library in-place,
1.4.2 are declared in a namespace whose name appends ::experimental::ranges_v1 to a namespace defined in the C++ Standard Library, such as std or std::chrono,
1.4.3 are declared in a subnamespace of a namespace described in the previous bullet, whose name is not the same as an existing subnamespace of namespace std,
1.4.4 are declared in namespace std::experimental::ranges::v1.

[Example: This TS does not define std::experimental::ranges_v1::chrono because the C++ Standard Library defines std::chrono. This paper does not define std::pmr::experimental::ranges_v1 because the C++ Standard Library does not define std::pmr. —end example]

[Editor’s note: The following text is new to this document. It is taken from the Library Fundamentals 2 TS and edited to reflect the fact that much of this document is suggesting parallel constrained facilities that are specified as diffs against the existing unconstrained facilities in namespace std.]

2 The International Standard, ISO/IEC 14882, together with N4377 (the Concepts TS), provide important context and specification for this paper. This document is written as a set of changes against ISO/IEC 14882. In places, this document suggests changes to be made to components in namespace std in-place. In other places, entire chapters and sections are copied verbatim and modified so as to define similar but different components in namespace std::experimental::ranges::v1. In those cases, effort was made to keep chapter and section numbers the same as in ISO/IEC 14882 for the sake of easy cross-referencing with the understanding that section numbers will change in the final draft.

3 Instructions to modify or add paragraphs are written as explicit instructions. Modifications made to existing text from the International Standard use underlining to represent added text and strikethrough to represent deleted text. Text in underline is used to denote text that was added since the last time the wording in question was reviewed, and strikethrough denotes text removed.
This paper assumes that the contents of the `std::experimental::ranges_v1::v1` namespace will become a new constrained version of the C++ Standard Library that will be delivered alongside the existing unconstrained version. The versioning mechanism to make this possible is yet to be determined.

Unless otherwise specified, references to other entities described in this document are assumed to be qualified with `std::experimental::ranges_v1::`, and references to entities described in the International Standard are assumed to be qualified with `std::`.

New headers are provided in the `<experimental/ranges_v1/>` directory. Where the new header has the same name as an existing header (e.g., `<experimental/ranges_v1/algorithm>`), the new header shall include the existing header as if by

```cpp
#include <algorithm>
```

### Table 1 — Ranges TS library headers

<table>
<thead>
<tr>
<th>Header</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;experimental/ranges_v1/algorithm&gt;</code></td>
<td><code>&lt;experimental/ranges_v1/random&gt;</code></td>
</tr>
<tr>
<td><code>&lt;experimental/ranges_v1/concepts&gt;</code></td>
<td><code>&lt;experimental/ranges_v1/tuple&gt;</code></td>
</tr>
<tr>
<td><code>&lt;experimental/ranges_v1/functional&gt;</code></td>
<td><code>&lt;experimental/ranges_v1/utility&gt;</code></td>
</tr>
<tr>
<td><code>&lt;experimental/ranges_v1/iterator&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
6 Statements

6.5 Iteration statements
6.5.4 The range-based for statement

[Editor’s note: Modify paragraph 1 to allow differently typed begin and end iterators.]

For a range-based for statement of the form

```cpp
for ( for-range-declaration : expression ) statement
```

let `range-init` be equivalent to the expression surrounded by parentheses

```cpp
(expression)
```

and for a range-based for statement of the form

```cpp
for ( for-range-declaration : braced-init-list ) statement
```

let `range-init` be equivalent to the `braced-init-list`. In each case, a range-based for statement is equivalent to

```cpp
{
    auto && _range = range-init;
    for ( auto __begin = begin-expr,
        __end = end-expr;
        __begin != __end;
        ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

```cpp
{
    auto && _range = range-init;
    auto __begin = begin-expr;
    auto __end = end-expr;
    for ( ; __begin != __end; ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

where `_range`, `_begin`, and `_end` are variables defined for exposition only, and `_RangeT` is the type of the expression, and `begin-expr` and `end-expr` are determined as follows:

1. if `_RangeT` is an array type, `begin-expr` and `end-expr` are `_range` and `_range + __bound`, respectively, where `__bound` is the array bound. If `_RangeT` is an array of unknown size or an array of incomplete type, the program is ill-formed;

2. if `_RangeT` is a class type, the `unqualified-ids begin` and `end` are looked up in the scope of class `_RangeT` as if by class member access lookup (3.4.5), and if either (or both) finds at least one declaration, `begin-expr` and `end-expr` are `_range.begin()` and `_range.end()`, respectively;

3. otherwise, `begin-expr` and `end-expr` are `begin(_range)` and `end(_range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (3.4.2). [Note: Ordinary unqualified lookup (3.4.1) is not performed. — end note]
Example:

```c
int array[5] = { 1, 2, 3, 4, 5 };  
for (int& x : array)  
  x *= 2;
```

— end example ]
17 Library introduction

17.5.1.3 Requirements

Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

(1.1) — Template arguments

(1.2) — Derived classes

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

The string and iostream components use an explicit representation of operations required of template arguments. They use a class template `char_traits` to define these constraints.

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

Requirements are stated in terms of concepts (N4377 [dcl.spec.concept]). Concepts are stated in terms of well-defined expressions that define valid terms of the types that satisfy the `requirements_concept`. For every set of well-defined expression requirements there is a `table_named concept` that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 25) that uses the well-defined expression requirements is described in terms of the valid expressions for its template type parameters.

Template argument requirements are sometimes referenced by name. See 17.5.2.1.

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

Required operations of any concept defined in this standard document need not be total functions; that is, some arguments to a required operation may result in the required semantics failing to be satisfied. [Example: The required `<` operator of the `StrictTotallyOrdered` concept (19.3.4) does not meet the semantic requirements of that concept when operating on NaNs. — end example] This does not affect whether a type satisfies the concept.

A declaration may explicitly impose requirements through its associated constraints ([14.10.2] in the Concepts TS). When the associated constraints refer to a concept (N4377 [dcl.spec.concept]), additional semantic requirements are imposed on the use of the declaration.

If the semantic requirements of a declaration are not satisfied at the point of use, the program is ill-formed, no diagnostic required.

17.5.2.1.5 Customization Point Objects

A customization point object is an object with a literal class type that interacts with user-defined types while enforcing semantic requirements on that interaction.

The type of a customization point object shall satisfy `Semiregular`.

All instances of a specific customization point object type shall be equal.

1) Although in some cases the code given is unambiguously the optimum implementation.
The type of a customization point object \( T \) shall satisfy \texttt{Function<const T, Args...>()} when the types of \texttt{Args...} meet the requirements specified in that customization point object’s definition. Otherwise, \( T \) shall not have a function call operator that participates in overload resolution.

Each customization point object type constrains its return type to satisfy a particular concept.

The library defines several named customization point objects. In every translation unit where such a name is defined, it shall refer to the same instance of the customization point object.

[\textit{Note:} Many of the customization points objects in the library evaluate function call expressions with an unqualified name which results in a call to a user-defined function found by argument dependent name lookup (3.4.2). To preclude such an expression resulting in a call to unconstrained functions with the same name in namespace \texttt{std}, customization point objects specify that lookup for these expressions is performed in a context that includes deleted overloads matching the signatures of overloads defined in namespace \texttt{std}. When the deleted overloads are viable, user-defined overloads must be more specialized (14.5.6.2) or more constrained (N4377 \[temp.constr.order\]) to be used by a customization point object. — end note]
19 Concepts library

[concepts.lib]

[Editor’s note: This chapter is inserted between the chapters [language.support] and [diagnostics]. All subsequence chapters should be renumbered as appropriate, but they aren’t here for the sake of simplicity.]

19.1 General

1 This Clause describes library components that C++ programs may use to perform compile-time validation of template parameters and perform function dispatch based on properties of types. The purpose of these concepts is to establish a foundation for equational reasoning in programs.

2 The following subclauses describe core language concepts, comparison concepts, object concepts, and function concepts as summarized in Table 2.

Table 2 — Fundamental concepts library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.2 Core language concepts</td>
<td>&lt;experimental/ranges-v1/concepts&gt;</td>
</tr>
<tr>
<td>19.3 Comparison concepts</td>
<td></td>
</tr>
<tr>
<td>19.4 Object concepts</td>
<td></td>
</tr>
<tr>
<td>19.5 Function concepts</td>
<td></td>
</tr>
</tbody>
</table>

3 In this Clause, CamelCase identifiers ending with “Type” denote alias templates.

4 The concepts in this Clause are defined in the namespace std::experimental::ranges-v1::v1.

19.1.1 Equality Preservation

1 An expression is equality preserving if, given equal inputs, the expression results in equal outputs. The inputs to an expression is the set of the expression’s operands. The output of an expression is the expression’s result and all operands modified by the expression. [ Note: Not all input values must be valid for a given expression; e.g., for integers a and b, the expression a / b is not well-defined when b is 0. This does not preclude the expression a / b being equality preserving. — end note ]

2 A regular function is a function that is equality preserving, i.e., a function that returns equal output when passed equal input. A regular function that returns a value may copy or move the returned object, or may return a reference. [ Note: Regular functions may have side effects. — end note ]

3 Expressions required by this specification to be equality preserving are further required to be stable: two evaluations of such an expression with the same input objects must have equal outputs absent any explicit intervening modification of those input objects. [ Note: This requirement allows generic code to reason about the current values of objects based on knowledge of the prior values as observed via equality preserving expressions. It effectively forbids spontaneous changes to an object, changes to an object from another thread of execution, changes to an object as side effects of non-modifying expressions, and changes to an object as side effects of modifying a distinct object if those changes could be observable to a library function via an equality preserving expression that is required to be valid for that object. — end note ]

4 Expressions declared in a requires-expression in this document are required to be equality preserving, except for those annotated with the comment “not required to be equality preserving.” An expression so annotated may be equality preserving, but is not required to be so.
An expression that may alter the value of one or more of its inputs in a manner observable to equality preserving expressions is said to modify those inputs. This document uses a notational convention to specify which expressions declared in a requires-expression modify which inputs: except where otherwise specified, an expression operand that is a non-constant lvalue or rvalue may be modified. Operands that are constant lvalues or rvalues must not be modified.

Where a requires-expression declares an expression that is non-modifying for some constant lvalue operand, additional variants of that expression that accept a non-constant lvalue or (possibly constant) rvalue for the given operand are also required except where such an expression variant is explicitly required with differing semantics. Such implicit expression variants must meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of these implicit expression variants is unspecified.

[Editor’s note: The motivation for this relaxation of syntactic checking is to avoid an exponential blow-up in the concept definitions and in compile times that would be required to check every permutation of cv-qualification and value category for expression operands.]

[Example:

template <class T>
concept bool C() {
    return requires (T a, T b, const T c, const T d) {
        c == d;  // #1
        a = std::move(b); // #2
        a = c;       // #3
    };
}

Expression #1 does not modify either of its operands, #2 modifies both of its operands, and #3 modifies only its first operand a.

Expression #1 implicitly requires additional expression variants that meet the requirements for c == d (including non-modification), as if the expressions

\[
\begin{align*}
    a == d; & \quad a == b; & \quad a == \text{move}(b); & \quad a == d; \\
    c == a; & \quad c == \text{move}(a); & \quad c == \text{move}(d); \\
    \text{move}(a) == d; & \quad \text{move}(a) == b; & \quad \text{move}(a) == \text{move}(b); & \quad \text{move}(a) == \text{move}(d); \\
    \text{move}(c) == b; & \quad \text{move}(c) == \text{move}(b); & \quad \text{move}(c) == d; & \quad \text{move}(c) == \text{move}(d);
\end{align*}
\]

had been declared as well.

Expression #3 implicitly requires additional expression variants that meet the requirements for a == c (including non-modification of the second operand), as if the expressions a == b and a == move(c) had been declared. Expression #3 does not implicitly require an expression variant with a non-constant rvalue second operand, since expression #2 already specifies exactly such an expression explicitly. —end example]

[Example: The following type T meets the explicitly stated syntactic requirements of concept C above but does not meet the additional implicit requirements:

\[
\begin{align*}
\text{struct T} \\
& \text{bool operator==}(\text{const T}&) \text{ const } \{ \text{return true; } \} \\
& \text{bool operator==}(\text{T}&) = \text{delete}; \\
\end{align*}
\]

T fails to meet the implicit requirements of C, so C<T>() is not satisfied. Since implementations are not required to validate the syntax of implicit requirements, it is unspecified whether or not an implementation diagnoses as ill-formed a program which requires C<T>(). —end example]

§ 19.1.1
19.2 Core language concepts

19.2.1 In general

This section contains the definition of concepts corresponding to language features. These concepts express relationships between types, type classifications, and fundamental type properties.

19.2.2 Concept Same

```
template <class T, class U>
concept bool Same() {
    return see below;
}
```

Same<T, U>() is satisfied if and only if T and U denote the same type.

Remarks: For the purposes of constraint checking, Same<T, U>() implies Same<U, T>().

19.2.3 Concept DerivedFrom

```
template <class T, class U>
concept bool DerivedFrom() {
    return see below;
}
```

DerivedFrom<T, U>() is satisfied if and only if is_base_of<U, T>::value is true.

19.2.4 ConceptConvertibleTo

```
template <class T, class U>
concept boolConvertibleTo() {
    return see below;
}
```

ConvertibleTo<T, U>() is satisfied if and only if is_convertible<T, U>::value is true.

19.2.5 Concept Common

```
template <class T, class U>
using CommonType = common_type_t<T, U>;
```

template <class T, class U>
concept bool Common() {
    return requires (T t, U u) {
        typename CommonType common_type_t<T, U>;
        typename CommonType common_type_t<U, T>;
        requires Same<CommonType common_type_t<U, T>, CommonType common_type_t<T, U>>();
        CommonType common_type_t<T, U>(std::forward<T>(t));
        CommonType common_type_t<T, U>(std::forward<U>(u));
    };
}
```

Let C be CommonType common_type_t<T, U>. Let t1 and t2 be objects of type T, and u1 and u2 be objects of type U. Common<T, U>() is satisfied if and only if

1. C(t1) equals C(t2) if and only if t1 equals t2.
2. C(u1) equals C(u2) if and only if u1 equals u2.

§ 19.2.5
Note: Users are free to specialize common_type when at least one parameter is a user-defined type. Those specializations are considered by the Common concept. — end note]

19.2.6 Concept Integral [concepts.lib.corelang.integral]

```
template <class T>
concept bool Integral() {
    return is_integral<T>::value;
}
```

19.2.7 Concept SignedIntegral [concepts.lib.corelang.signedintegral]

```
template <class T>
concept bool SignedIntegral() {
    return Integral<T>() && is_signed<T>::value;
}
```

[Note: SignedIntegral<T>() may be satisfied even for types that are not signed integral types (3.9.1); for example, char. — end note]

19.2.8 Concept UnsignedIntegral [concepts.lib.corelang.unsignedintegral]

```
template <class T>
concept bool UnsignedIntegral() {
    return Integral<T>() && !SignedIntegral<T>();
}
```

[Note: UnsignedIntegral<T>() may be satisfied even for types that are not unsigned integral types (3.9.1); for example, char. — end note]

19.2.9 Concept Assignable [concepts.lib.corelang.assignable]

```
template <class T, class U = T>
concept bool Assignable() {
    return requires(T&& a, U&& b) {
        { std::forward<T>(a) = std::forward<U>(b) } -> Same<T&>;
    };
}
```

Let t be an lvalue of type T. If U is an lvalue reference type, let v be a lvalue of type U; otherwise, let v be an rvalue of type U. Then Assignable<T, U>() is satisfied if and only if

\[ \text{std::addressof}(t = v) == \text{std::addressof}(t). \] (1.1)

19.2.10 Concept Swappable [concepts.lib.corelang.swappable]

[Editor’s note: Remove subclause [swappable.requirements]. Replace references to [swappable.requirements] with [concepts.lib.corelang.swappable].]

```
template <class T>
concept bool Swappable() {
    return requires(T&& a, T&& b) {
        (void)ranges::swap(std::forward<T>(a), std::forward<T>(b));
    };
}
```

template <class T, class U>
concept bool Swappable() {
    return Swappable<T>() &&
Swappable<T, U>() &k
Common<T, U>() &k
requires(T& t, U& u) {
  (void) ranges::swap(std::forward<T>(t), std::forward<U>(u));
  (void) ranges::swap(std::forward<U>(u), std::forward<T>(t));
};

[Note: The casts to void in the required expressions indicate that the value—if any—of the call to
swap does not participate in the semantics. Callers are effectively forbidden to rely on the return type
or value of a call to swap, and user customizations of swap need not return equal values when given
equal operands. —end note]

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 \( \text{Swappable}<T, U>() \) is satisfied. \( \text{Swappable}<T, U>() \) is satisfied if and only if:

- given distinct objects \( tt \) equal to \( t \) and \( uu \) equal to \( u \), after evaluating
  either \( \text{ranges::swap}(t, u) \) or \( \text{ranges::swap}(u, t) \), \( tt \) is equal to \( u \) and \( uu \) is equal to \( t \).

- the requires clause above is evaluated in the context described below, and
- these expressions have the following effects:
  - the object referred to by \( t \) has the value originally held by \( u \) and
  - the object referred to by \( u \) has the value originally held by \( t \).

The context in which the requires clause is evaluated shall ensure that a binary non-member function
named “\( \text{swap} \)” is selected via overload resolution (13.3) on a candidate set that includes:

- the two \( \text{swap} \) function templates defined in \( \langle \text{utility} \rangle \) (20.2) and
- the lookup set produced by argument-dependent lookup (3.4.2).

[Note: If \( T \) and \( U \) are both fundamental types or arrays of fundamental types and the declarations
from the header \( \langle \text{utility} \rangle \) are in scope, the overall lookup set described above is equivalent to that
of the qualified name lookup applied to the expression \( \text{std::swap}(t, u) \) or \( \text{std::swap}(u, t) \) as
appropriate. —end note]

[Note: It is unspecified whether a library component that has a swappable requirement includes the
header \( \langle \text{utility} \rangle \) to ensure an appropriate evaluation context. —end note]

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 \( \text{swap} \) 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 U>
void value_swap(T&& t, U&& u) {
  using std::experimental::ranges::swap;
  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 T>
void lv_swap(T& t1, T& t2) {
  using std::experimental::ranges::swap;
```
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) {     
        std::experimental::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 };             
    value_swap(a1, proxy(a2));                
    assert(a1.m == -5 && a2.m == 5);          
}                                               

—end example]                                 

19.3 Comparison concepts [concepts.lib.compare]

19.3.1 In general [concepts.lib.compare.general]

This section describes concepts that establish relationships and orderings on values of possibly differing
object types.

19.3.2 Concept Boolean [concepts.lib.compare.boolean]

The Boolean concept describes the requirements on a type that is usable in Boolean contexts.

template <class B>
concept bool Boolean() {                     
    return MoveConstructible<B>() && // (see 19.4.4)
    requires(const B b1, const B b2, const bool a) {
        bool(b1);                           
        { b1 } -> bool;                     
        bool(!b1);                          
        { !b1 } -> bool;                    
        { b1 && b2 } -> Same<bool>;         
        { b1 && a } -> Same<bool>;          
        { a && b1 } -> Same<bool>;          
        { b1 || b2 } -> Same<bool>;         
        { b1 || a } -> Same<bool>;          
        { a || b1 } -> Same<bool>;          
        { b1 == b2 } -> bool;               
        { b1 != b2 } -> bool;               
        { b1 == a } -> bool;                
        { a == b1 } -> bool;                
        { b1 != a } -> bool;

§ 19.3.2 13
Given values \( b_1 \) and \( b_2 \) of type \( B \), then \( \text{Boolean}<B>() \) is satisfied if and only if

\[
\begin{align*}
(2.1) & \quad \text{bool}(b_1) = [\text{bool}(x) \{ \text{return } x; \}](b_1). \\
(2.2) & \quad \text{bool}(b_1) = !\text{bool}(!b_1). \\
(2.3) & \quad (b_1 && b_2), (b_1 && \text{bool}(b_2)), \text{and } \text{bool}(b_1) && b_2 \text{ are all equal to } \text{bool}(b_1) && \text{bool}(b_2), \text{ and have the same short-circuit evaluation.} \\
(2.4) & \quad (b_1 || b_2), (b_1 || \text{bool}(b_2)), \text{and } \text{bool}(b_1) || b_2 \text{ are all equal to } \text{bool}(b_1) || \text{bool}(b_2), \text{ and have the same short-circuit evaluation.} \\
(2.5) & \quad \text{bool}(b_1 == b_2), \text{bool}(b_1 == \text{bool}(b_2)), \text{and } \text{bool}(\text{bool}(b_1) == b_2) \text{ are all equal to } \text{bool}(b_1) == \text{bool}(b_2). \\
(2.6) & \quad \text{bool}(b_1 != b_2), \text{bool}(b_1 != \text{bool}(b_2)), \text{and } \text{bool}(\text{bool}(b_1) != b_2) \text{ are all equal to } \text{bool}(b_1) != \text{bool}(b_2). 
\end{align*}
\]

\[ \text{Example: The types bool, std::true_type, and std::bitset<}N>::reference are Boolean types. Pointers, smart pointers, and types with explicit conversions to bool are not Boolean types. — end example] \]

19.3.3 Concept EqualityComparable

[concepts.lib.compare.equalitycomparable]

[Editor’s note: Remove table [equalitycomparable] in [utility.arg.requirements]. Replace references to [equalitycomparable] with [concepts.lib.compare.equalitycomparable].]

```cpp
template <class T>
concept bool EqualityComparable() {
    return requires(const T a, const T b) {
        {a == b} -> Boolean;
        {a != b} -> Boolean;
    };
}
```

\begin{align*}
(1.1) & \quad \text{bool}(a == a). \\
(1.2) & \quad \text{bool}(a == b) \text{ if and only if } a \text{ is equal to } b. \\
(1.3) & \quad \text{bool}(a != b) == !\text{bool}(a == b). \\
\end{align*}

[Note: The requirement that the expression \( a == b \) is equality preserving implies that \( == \) is transitive and symmetric. — end note]

```cpp
template <class T, class U>
concept bool EqualityComparable() {
    return Common<T, U>() &
        EqualityComparable<T>() &
        EqualityComparable<U>() &
        EqualityComparable<CommonType<common_type_t<T, U>>()>() &
        requires(const T a, const U b) {
            {a == b} -> Boolean;
            {b == a} -> Boolean;
            {a != b} -> Boolean;
            {b != a} -> Boolean;
        };
}
```

§ 19.3.3
Let a be an object of type T, b be an object of type U, and C be CommonType\textsubscript{common\_type\_t<T, U>}. Then EqualityComparable\textsubscript{T, U>() is satisfied if and only if

\[\text{(3.1)}\] 
\[\text{bool}(a == b) == \text{bool}(C(a) == C(b)).\]

\[\text{(3.2)}\] 
\[\text{bool}(a != b) == \text{bool}(C(a) != C(b)).\]

\[\text{(3.3)}\] 
\[\text{bool}(b == a) == \text{bool}(C(b) == C(a)).\]

\[\text{(3.4)}\] 
\[\text{bool}(b != a) == \text{bool}(C(b) != C(a)).\]

### 19.3.4 Concept StrictTotallyOrdered

[concepts.lib.compare.stricttotallyordered]

[Editor's note: Remove table [lessthancomparable] in [utility.arg.requirements]. Replace uses of LessThanComparable with StrictTotallyOrdered (acknowledging that this is a breaking change that makes type requirements stricter). Replace references to [lessthancomparable] with references to [concepts.lib.compare.stricttotallyordered]]

```cpp
19.3.4 Concept StrictTotallyOrdered

template <class T>
concept bool StrictTotallyOrdered() {
    return EqualityComparable<T>() &&
    requires (const T a, const T b) {
        { a < b } -> Boolean;
        { a > b } -> Boolean;
        { a <= b } -> Boolean;
        { a >= b } -> Boolean;
    }
}
```

Let a, b, and c be objects of type T. Then StrictTotallyOrdered\textsubscript{T>() is satisfied if and only if

\[\text{(1.1)}\] 
\[\text{Exactly one of bool}(a < b), bool(b < a), or bool(a == b) is true.}\]

\[\text{(1.2)}\] 
\[\text{If bool}(a < b) and bool(b < c), then bool(a < c).}\]

\[\text{(1.3)}\] 
\[\text{bool}(a > b) == bool(b < a).}\]

\[\text{(1.4)}\] 
\[\text{bool}(a <= b) == !bool(b < a).}\]

\[\text{(1.5)}\] 
\[\text{bool}(a >= b) == !bool(a < b).}\]

[Note: Not all arguments will be well-formed for a given type. For example, NaN is not a well-formed floating point value, and many types’ moved-from states are not well-formed. This does not mean that the type does not satisfy StrictTotallyOrdered. — end note]

```cpp
template <class T, class U>
concept bool StrictTotallyOrdered() {
    return Common\textsubscript{T, U>() &&
    StrictTotallyOrdered\textsubscript{T>() &&
    StrictTotallyOrdered\textsubscript{U>() &&
    StrictTotallyOrdered\textsubscript{CommonType\_common\_type\_t<T, U>}() &&
    EqualityComparable\textsubscript{T, U>() &&
    requires (const T t, const U u) {
        { t < u } -> Boolean;
        { t > u } -> Boolean;
        { t <= u } -> Boolean;
        { t >= u } -> Boolean;
        { u < t } -> Boolean;
        { u > t } -> Boolean;
        { u <= t } -> Boolean;
        { u >= t } -> Boolean;
    }
}
```

§ 19.3.4
Let \( t \) be an object of type \( T \), \( u \) be an object of type \( U \), and \( C \) be \( \text{CommonType}<T, U> \). Then \( \text{StrictTotallyOrdered}<T, U>() \) is satisfied if and only if

\[
\begin{align*}
(3.1) & \quad \text{bool}(t < u) == \text{bool}(C(t) < C(u)). \\
(3.2) & \quad \text{bool}(t > u) == \text{bool}(C(t) > C(u)). \\
(3.3) & \quad \text{bool}(t <= u) == \text{bool}(C(t) <= C(u)). \\
(3.4) & \quad \text{bool}(t >= u) == \text{bool}(C(t) >= C(u)). \\
(3.5) & \quad \text{bool}(u < t) == \text{bool}(C(u) < C(t)). \\
(3.6) & \quad \text{bool}(u > t) == \text{bool}(C(u) > C(t)). \\
(3.7) & \quad \text{bool}(u <= t) == \text{bool}(C(u) <= C(t)). \\
(3.8) & \quad \text{bool}(u >= t) == \text{bool}(C(u) >= C(t)). 
\end{align*}
\]

19.4 Object concepts

This section defines concepts that describe the basis of the value-oriented programming style on which the library is based. [Editor’s note: These concepts reuse many of the names of concepts that traditionally describe features of types to describe families of object types.]

19.4.1 Concept Destructible

[concepts.lib.object.destructible] [concepts.lib.object]

The Destructible concept is the base of the hierarchy of object concepts. It describes properties that all such object types have in common.

```cpp
template <class T>
concept bool Destructible() {
    return requires (T t, const T ct, T* p) {
        { t.~T() } noexcept;
        { &t } -> Same<T*>; // not required to be equality preserving
        { &ct } -> Same<const T*>; // not required to be equality preserving
        delete p;
        delete[] p;
    };
}
```

The expression requirement \&ct does not require implicit expression variants.

Given a (possibly const) lvalue \( t \) of type \( T \) and pointer \( p \) of type \( T* \), \( \text{Destructible}<T>() \) is satisfied if and only if

\[
\begin{align*}
(3.1) & \quad \text{After evaluating the expression } t.~T(), \text{ delete } p, \text{ or delete}[] p, \text{ all resources owned by the denoted object(s) are reclaimed.} \\
(3.2) & \quad \&t == \text{std::addressof}(t). \\
(3.3) & \quad \text{The expression } \&t \text{ is non-modifying.}
\end{align*}
\]

19.4.2 Concept Constructible

[concepts.lib.object.constructible]

The Constructible concept is used to constrain the type of a variable to be either an object type constructible from a given set of argument types, or a reference type that can be bound to those arguments.

```cpp
template <class T, class ...Args>
concept bool __ConstructibleObject = // exposition only
    Destructible<T>() && requires (Args&& ...args) {
        T(\text{std::forward<Args>}(\text{args})...); // not required to be equality preserving
    }
```
new T(std::forward<Args>(args)...) // not required to be equality preserving
};

template <class T, class ...Args>
concept bool __BindableReference = // exposition only
    is_reference<T>::value && requires (Args&& ...args) {
        T(std::forward<Args>(args)...);
    };

template <class T, class ...Args>
concept bool Constructible() {
    return __ConstructibleObject<T, Args...> ||
    __BindableReference<T, Args...>;
}

19.4.3 Concept DefaultConstructible [concepts.lib.object.defaultconstructible]
[Editor's note: Remove table [defaultconstructible] in [utility.arg.requirements]. Replace references to [defaultconstructible] with references to [concepts.lib.object.defaultconstructible].]

template <class T>
concept bool DefaultConstructible() {
    return Constructible<T>() &&
    requires (const size_t n) {
        new T[n]{}; // not required to be equality preserving
    };
}

[Note: The array allocation expression new T[n]{} implicitly requires that T has a non-explicit default constructor. —end note]

19.4.4 Concept MoveConstructible [concepts.lib.object.moveconstructible]
[Editor's note: Remove table [moveconstructible] in [utility.arg.requirements]. Replace references to [moveconstructible] with references to [concepts.lib.object.moveconstructible].]

template <class T>
concept bool MoveConstructible() {
    return Constructible<T, remove_cv_t<T>&&>() &&
    Convertible<remove_cv_t<T>&& T>{};
}

1 [Note: Let rv be an rvalue of type remove_cv_t<T>. Then MoveConstructible<T>() is satisfied if and only if

  (1.1) — After the definition T u = rv; u is equal to the value of rv before the construction.

  (1.2) — T{rv} or *new T{rv} is equal to the value of rv before the construction.

2 rv’s resulting state is unspecified. [Note: rv must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether rv has been moved from or not. — end note]

[Editor’s note: Ideally, MoveConstructible would include an array allocation requirement new T[1]{std::move(t)}. This is not currently possible since [expr.new]/19 requires an accessible default constructor even when all array elements are initialized.]

19.4.5 Concept CopyConstructible [concepts.lib.object.copyconstructible]
[Editor’s note: Remove table [copyconstructible] in [utility.arg.requirements]. Replace references to [copyconstructible] with references to [concepts.lib.object.copyconstructible].]
template <class T>
concept bool CopyConstructible() {
  return MoveConstructible<T>() &&
      Constructible<T, const remove_cv_t<T>&>() &&
     Convertible<remove_cv_t<T>&, T&>() &&
     Convertible<const remove_cv_t<T>&, T&>() &&
     Convertible<const remove_cv_t<T>&&, T>();
}

Let \( v \) be an lvalue of type (possibly const) \( \text{remove} \_\text{cv} \_t \text{T} \) or an rvalue of type const \( \text{remove} \_\text{cv} \_t \text{T} \). Then \( \text{CopyConstructible<T>()} \) is satisfied if and only if

\[(1.1) \quad \text{— After the definition} \quad \text{T} \ u = v; \quad \text{v is equal to} \quad u.\]
\[(1.2) \quad \text{— T(v) or *new T(v) is equal to} \quad v.\]

[Editor's note: Ideally, \text{CopyConstructible} would include an array allocation requirement \text{new T[1]}\{t\}. This is not currently possible since \text{expr.new}/19 requires an accessible default constructor even when all array elements are initialized.]

19.4.6 Concept Movable [concepts.lib.object.movable]

[Editor's note: Remove table [moveassignable] in [utility.arg.requirements]. Replace references to [move-assignable] with references to [concepts.lib.object.movable].]

template <class T>
concept bool Movable() {
  return MoveConstructible<T>() &&
         Assignable<T&, T&&>();
}

Let \( r v \) be an rvalue of type \( T \) and let \( t \) be an lvalue of type \( T \). Then \( \text{Movable<T>()} \) is satisfied if and only if

\[(1.1) \quad \text{— After the assignment} \quad t = r v; \quad t \text{ is equal to the value of} \quad r v \text{ before the assignment.} \]
\[(2) \quad r v \text{’s resulting state is unspecified.} \quad \text{[Note:} \quad r v \quad \text{must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether} \quad r v \quad \text{has been moved from or not. — end note]} \]

19.4.7 Concept Copyable [concepts.lib.object.copyable]

[Editor's note: Remove table [copyassignable] in [utility.arg.requirements]. Replace references to [copy-assignable] with references to [concepts.lib.object.copyable].]

template <class T>
concept bool Copyable() {
  return CopyConstructible<T>() &&
         Movable<T>() &&
         Assignable<T&, const T&>();
}

Let \( t \) be an lvalue of type \( T \), and \( v \) be an lvalue of type (possibly const) \( T \) or an rvalue of type const \( T \). Then \( \text{Copyable<T>()} \) is satisfied if and only if

\[(1.1) \quad \text{— After the assignment} \quad t = v; \quad t \text{ is equal to} \quad v.\]

19.4.8 Concept Semiregular [concepts.lib.object.semiregular]

template <class T>
concept bool Semiregular() {
  return Copyable<T>() &&
DefaultConstructible<T>();
}

[Note: The Semiregular concept is satisfied by types that behave similarly to built-in types like int, except that they may not be comparable with ==. — end note]

19.4.9 Concept Regular [concepts.lib.object.regular]

template <class T>
concept bool Regular() {
    return Semiregular<T>() &&
        EqualityComparable<T>();
}

[Note: The Regular concept is satisfied by types that behave similarly to built-in types like int and that are comparable with ==. — end note]

19.5 Function concepts [concepts.lib.functions]

19.5.1 In general [concepts.lib.functions.general]

The function concepts in this section describe the requirements on function objects (20.9) and their arguments.

19.5.2 Concept Function [concepts.lib.functions.function]

    template <class F, class...Args>
    using ResultType = result_of_t<F(Args...)>

    template <class F, class...Args>
    concept bool Function() {
        return CopyConstructible<F>() &&
            requires (F f, Args&&...args) {
                typename ResultType<F, Args...> result_of_t<F&>(Args...);
                { f(std::forward<Args>(args)...) } // not required to be equality preserving
                -> Same<ResultType<F, Args...> result_of_t<F&>(Args...)>;
            };
    }

[Note: Since the function call expression of Function is not required to be equality-preserving (19.1.1), a function that generates random numbers may satisfy Function. — end note]

19.5.3 Concept RegularFunction [concepts.lib.functions.regularfunction]

    template <class F, class...Args>
    concept bool RegularFunction() {
        return Function<F, Args...>();
    }

The function call expression of RegularFunction shall be equality-preserving (19.1.1). [Note: This requirement supersedes the annotation on the function call expression in the definition of Function. — end note]

[Note: A random number generator does not satisfy RegularFunction. — end note]

3 [Note: There is no syntactic difference between Function and RegularFunction. — end note]

19.5.4 Concept Predicate [concepts.lib.functions.predicate]

    template <class F, class...Args>

§ 19.5.4
concept bool Predicate() {
    return RegularFunction<F, Args...>() &&
    Boolean<ResultType<F, Args...>::result_of_t<F&>(Args...)>();
}

19.5.5 Concept Relation
[concepts.lib.functions.relation]
template <class R, class T>
concept bool Relation() {
    return Predicate<R, T, T>();
}

template <class R, class T, class U>
concept bool Relation() {
    return Relation<R, T, U>() &&
    Relation<R, CommonType::common_type_t<T, U>, U>() &&
    Predicate<R, T, U>() &&
    Predicate<R, U, T>();
}

Let \( r \) be any object of type \( R \), \( a \) be any object of type \( T \), \( b \) be any object of type \( U \), and \( C \) be CommonType::common_type_t<T, U>. Then Relation<R, T, U>() is satisfied if and only if

\[
\begin{align*}
(1.1) & \quad \text{bool}(r(a, b)) == \text{bool}(r(C(a), C(b))). \\
(1.2) & \quad \text{bool}(r(b, a)) == \text{bool}(r(C(b), C(a))).
\end{align*}
\]

19.5.6 Concept StrictWeakOrder
[concepts.lib.functions.strictweakorder]
template <class R, class T>
concept bool StrictWeakOrder() {
    return Relation<R, T>();
}

template <class R, class T, class U>
concept bool StrictWeakOrder() {
    return Relation<R, T, U>();
}

A Relation satisfies StrictWeakOrder if and only if it imposes a strict weak ordering on its arguments.

[Editor's note: Copied verbatim from [alg.sorting].]

The term strict refers to the requirement of an irreflexive relation \( \neg \text{comp}(x, x) \) for all \( x \), and the term weak to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define equiv(a, b) as \( \neg \text{comp}(a, b) \land \neg \text{comp}(b, a) \), then the requirements are that comp and equiv both be transitive relations:

\[
\begin{align*}
(2.1) & \quad \text{comp}(a, b) \land \text{comp}(b, c) \Rightarrow \text{comp}(a, c) \\
(2.2) & \quad \text{equiv}(a, b) \land \text{equiv}(b, c) \Rightarrow \text{equiv}(a, c) \ [\text{Note: Under these conditions, it can be shown that}]
\end{align*}
\]

\[
\begin{align*}
(2.2.1) & \quad \text{equiv} \text{ is an equivalence relation} \\
(2.2.2) & \quad \text{comp induces a well-defined relation on the equivalence classes determined by equiv} \\
(2.2.3) & \quad \text{The induced relation is a strict total ordering.} \ [\text{end note}]
\end{align*}
\]

\[\text{§ 19.5.6}\]
20 General utilities library

20.2 Utility components

Header <utility> synopsis

1 Change the <utility> synopsis as follows:

```cpp
// ... as before ..

// 20.2.2, swap:
template<class T>
void swap(T& a, T& b) noexcept(see below);

template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

// ... as before ..
```

2 Header <experimental/ranges_v1/utility> synopsis

```cpp
namespace std {
    namespace experimental {
        namespace ranges_v1 { inline namespace v1 {
            // 20.2.2, swap:
            using std::swap;

            namespace {
                constexpr unspecified swap = unspecified;
            }

            // 20.2.3, exchange:
template <class Movable T, class U=T>
    requires Assignable<T&, U>()
    T exchange(T& obj, U& new_val);

            // 20.15.2, struct with named accessors
    template <class T>
    concept bool TagSpecifier() {
        return see below;
    }

    template <class F>
    concept bool TaggedType() {
        return see below;
    }

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

            // 20.15.3, tagged specialized algorithms
    template <class Base, class... Tags>
    requires Swappable<Base>()
    void swap(tagged<Base, Tag...>& x, tagged<Base, Tag...>& y) noexcept(see below);
        }
    }
}
```
// 20.15.5, tagged pairs
template <TaggedType T1, TaggedType T2> using tagged_pair = see below;

template <TagSpecifier Tag1, TagSpecifier Tag2, class T1, class T2>
constexpr tagged_pair<Tag1(V1), Tag2(V2)> make_tagged_pair(T1&& x, T2&& y);

namespace std {
    // 20.15.4, tuple-like access to tagged
    template <class Base, class... Tags>
    struct tuple_size<experimental::ranges_v1::tagged<Base, Tags...>>;

    template <size_t N, class Base, class... Tags>
    struct tuple_element<N, experimental::ranges_v1::tagged<Base, Tags...>>;
}

Any entities declared or defined directly in namespace std in header <utility> that are not already defined in namespace std::experimental::ranges_v1 in header <experimental/ranges_v1/utility> are imported with using-declarations (7.3.3). [Example:

namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
    using std::pair;
    using std::make_pair;
    // ... others
}}}
—end example]

Any nested namespaces defined directly in namespace std in header <utility> that are not already defined in namespace std::experimental::ranges_v1 in header <experimental/ranges_v1/utility> are aliased with a namespace-alias-definition (7.3.2). [Example:

namespace std { namespace experimental { namespace ranges_v1 {
    namespace rel_ops = std::rel_ops;
}}}
—end example]

20.2.2 swap [utility.swap]

The name swap denotes a customization point object (17.5.2.1.5). The effect of the expression ranges::swap(E1, E2) for some expressions E1 and E2 is equivalent to:

(1.1) — (void)swap(E1, E2), with overload resolution performed in a context that includes the declarations

    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 denoted by E1 and E2, the program is ill-formed with no diagnostic required.

(1.2) — Otherwise, (void)swap_ranges(E1, E2) if E1 and E2 are lvalues of array types (3.9.2) of 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)).
[Editor's note: This formulation intentionally allows swapping arrays with identical extent and differing element types, but only when swapping the element types is well-defined. Swapping arrays of int and double continues to be unsound, but Swappable<T&, U&>() implies Swappable<T(N), U(N)>.]

(1.3) — Otherwise, if E1 and E2 are lvalues of the same type T which meets the syntactic requirements of Movable<T>(), exchanges the denoted 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 Movable<T>() is not satisfied, the program is ill-formed with no diagnostic required. ranges::swap(E1, E2) has type void.

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

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

[Editor's note: The following edits to the two swap overloads are to be applied to the versions in namespace std.]

template<class T>
void swap(T& a, T& b) noexcept(see below);

3 Remarks: The expression inside noexcept is equivalent to:

is_nothrow_move_constructible<T>::value &&
is_nothrow_move_assignable<T>::value

4 Remarks: A library implementor is free to omit the requires clause so long as this function does not participate in overload resolution if the following is false

is_move_constructible<T>::value &&
is_move_assignable<T>::value &&
is_destructible<T>::value

5 Requires: Type T shall be MoveConstructible (Table 20) and MoveAssignable (Table 22).

6 Effects: Exchanges values stored in two locations.

template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

7 Requires: a[i] shall be swappable with (19.2.10) b[i] for all i in the range [0,N).

8 Effects: swap_ranges(a, a + N, b)

20.2.3 exchange

[utility.exchange]

template <class Movable T, class U=T>
requires Assignable<T&, U>()
T exchange(T& obj, U& new_val);

1 Effects: Equivalent to:

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

§ 20.2.3
20.9 Function objects

Header `<experimental/ranges_v1/functional>` synopsis

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
  template <class T = void>
  requires EqualityComparable<T>() || Same<T, void>()
  struct equal_to;

  template <class T = void>
  requires EqualityComparable<T>() || Same<T, void>()
  struct not_equal_to;

  template <class T = void>
  requires StrictTotallyOrdered<T>() || Same<T, void>()
  struct greater;

  template <class T = void>
  requires StrictTotallyOrdered<T>() || Same<T, void>()
  struct less;

  template <class T = void>
  requires StrictTotallyOrdered<T>() || Same<T, void>()
  struct greater_equal;

  template <class T = void>
  requires StrictTotallyOrdered<T>() || Same<T, void>()
  struct less_equal;

  struct identity;
}}}
```

3 Any entities declared or defined directly in namespace `std` in header `<functional>` that are not already defined in namespace `std::experimental::ranges_v1` in header `<experimental/ranges_v1/functional>` are imported with `using-declarations` (7.3.3). [Example:

```cpp
namespace std { namespace experimental { namespace ranges_v1 {
  using std::reference_wrapper;
  using std::ref;
  // ... others
}}}
```

—end example]

4 Any nested namespaces defined directly in namespace `std` in header `<functional>` that are not already defined in namespace `std::experimental::ranges_v1` in header `<experimental/ranges_v1/functional>` are aliased with a `namespace-alias-definition` (7.3.2). [Example:

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
  namespace placeholders = std::placeholders;
}}}
```

—end example]

20.9.6 Comparisons

The library provides basic function object classes for all of the comparison operators in the language (5.9, 5.10).
template <class T = void>
  requires EqualityComparable<T>() || Same<T, void>()
struct equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};

operator() returns \(x == y\).

template <class T = void>
  requires EqualityComparable<T>() || Same<T, void>()
struct not_equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
  typedef T first_argument_type;
  typedef T second_argument_type;
  typedef bool result_type;
};

operator() returns \(x != y\).

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

operator() returns \(x > y\).

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

operator() returns \(x < y\).

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

operator() returns \(x >= y\).

template <class T = void>
  requires StrictTotallyOrdered<T>() || Same<T, void>()
§ 20.9.6

25
struct less_equal {
    constexpr bool operator()(const T& x, const T& y) const;
    typedef T first_argument_type;
    typedef T second_argument_type;
    typedef bool result_type;
};

operator() returns \( x \leq y \).

template <> struct equal_to<void> {
    template <class T, class U>
    requires EqualityComparable<T, U>()
    constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) == std::forward<U>(u));

    typedef unspecified is_transparent;
};

operator() returns \( \text{std::forward}(t) == \text{std::forward}(u) \).

template <> struct not_equal_to<void> {
    template <class T, class U>
    requires EqualityComparable<T, U>()
    constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) != std::forward<U>(u));

    typedef unspecified is_transparent;
};

operator() returns \( \text{std::forward}(t) \neq \text{std::forward}(u) \).

template <> struct greater<void> {
    template <class T, class U>
    requires StrictTotallyOrdered<T, U>()
    constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) > std::forward<U>(u));

    typedef unspecified is_transparent;
};

operator() returns \( \text{std::forward}(t) > \text{std::forward}(u) \).

template <> struct less<void> {
    template <class T, class U>
    requires StrictTotallyOrdered<T, U>()
    constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) < std::forward<U>(u));

    typedef unspecified is_transparent;
};

operator() returns \( \text{std::forward}(t) < \text{std::forward}(u) \).

template <> struct greater_equal<void> {
    template <class T, class U>
    requires StrictTotallyOrdered<T, U>()
    constexpr auto operator()(T&& t, U&& u) const

§ 20.9.6
```cpp
decltype(std::forward<T>(t) >= std::forward<U>(u));

typedef unspecified is_transparent;
}

operator() returns std::forward<T>(t) >= std::forward<U>(u).

```

```
12
operator() returns std::forward<T>(t) >= std::forward<U>(u).

```

```
13
operator() returns std::forward<T>(t) <= std::forward<U>(u).

```

```
14
For templates greater, less, greater_equal, and less_equal, the specializations for any pointer type yield a total order, even if the built-in operators <, >, <=, >= do not.

```

[Editor's note: After subsection 20.9.12 [unord.hash] add the following subsection:

20.9.13 Class identity [func.identity]

struct identity {
    template <class T>
    constexpr T&& operator()(T&& t) const noexcept;

typedef unspecified is_transparent;
};

operator() returns std::forward<T>(t).

[Editor's note: REVIEW: From Stephan T. Lavavej: "[This] identity functor, being a non-template, clashes with any attempt to provide identity<T>::type." <Insert bikeshed naming discussion here>.]

20.15 Tagged tuple-like types [taggedtup]

20.15.1 In general [taggedtup.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 20.4.2.6).

[Editor's note: This type exists so that the algorithms can return pair- and tuple-like objects with named accessors, as requested by LEWG. Rather than create a bunch of one-off class types with no relation to pair and tuple, I opted instead to create a general utility. I’ll note that this functionality can be merged into pair and tuple directly with minimal breakage, but I opted for now to keep it separate.]

20.15.2 Class template tagged [taggedtup.tagged]

Class template tagged augments a tuple-like class type (e.g., pair (20.3), tuple (20.4)) by giving it named accessors. It is used to define the alias templates tagged_pair (20.15.5) and tagged_tuple (20.15.6).

In the class synopsis below, let i be in the range [0, sizeof...(Tags)) in order and Ti be the i\textsuperscript{th} type in Tags, where indexing is zero-based.
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() & { return get<N>(static_cast<D&>(*this)); }
    constexpr decltype(auto) name() && { return get<N>(static_cast<D&&>(*this)); }
    constexpr decltype(auto) name() const & { return get<N>(static_cast<const D&>(*this)); }
};
```

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 $F$.

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

Remarks: The expression in the noexcept is equivalent to:

```cpp
is_nothrow_constructible<Base, Other>::value
```

Effects: Initializes Base with static_construct<Base, Other>&&(that).

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

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

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

Remarks: The expression in the noexcept is equivalent to:

```cpp
is_nothrow_assignable<Base&, Other>::value
```

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

Returns: *this.

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

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

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

  Remarks: The expression in the noexcept is equivalent to:
            is_nothrow_assignable<Base&, U>::value

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

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

  Remarks: The expression in the noexcept is equivalent to:
            noexcept(swap(declval<Base&>(), declval<Base&>()))

  Effects: Calls swap on the result of applying static_cast to *this and that.
  Throws: Nothing unless the call to swap on the Base sub-objects throws.

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

  Remarks: The expression in the noexcept is equivalent to:
            noexcept(lhs.swap(rhs))

  Effects: lhs.swap(rhs).
  Throws: Nothing unless the call to lhs.swap(rhs) throws.

20.15.3 tagged specialized algorithms

20.15.4 Tuple-like access to tagged

20.15.5 Alias template tagged_pair

§ 20.15.5
using tagged_pair = tagged<pair<TAGELEM(T1), TAGELEM(T2)>,
    TAGSPEC(T1), TAGSPEC(T2)>;
}}}

1 [Example:
   // See 25.1:
   tagged_pair<tag::min(int), tag::max(int)> p{0, 1};
   assert(&p.min() == &p.first);
   assert(&p.max() == &p.second);
   —end example]

20.15.5.1 Tagged pair creation functions
   [tagged.pairs.creation]
   // defined in header <experimental/ranges++/utility>

namespace std { namespace experimental { namespace ranges_v1 {
    template <TagSpecifier Tag1, TagSpecifier Tag2, class T1, class T2>
    constexpr tagged_pair<Tag1(V1), Tag2(V2)> make_tagged_pair(T1&& x, T2&& y);
}}

1 Returns: tagged_pair<Tag1(V1), Tag2(V2)>>(std::forward<T1>(x), std::forward<T2>(y));
   where V1 and V2 are determined as follows: Let Ui be decay_t<Ti> for each Ti. Then each Vi is X&
   if Ui equals reference_wrapper<X>, otherwise Vi is Ui.

2 [Example: In place of:
   return tagged_pair<tag::min(int), tag::max(double)>(5, 3.1415926);
   // explicit types
   a C++ program may contain:
   return make_tagged_pair<tag::min, tag::max>(5, 3.1415926);
   // types are deduced
   —end example]

20.15.6 Alias template tagged_tuple
   [tagged.tuple]
   Header <experimental/ranges++/tuple> synopsis

namespace std { namespace experimental { namespace ranges_v1 {
    template <TaggedType... Types>
    using tagged_tuple = tagged<tuple<TAGELEM(Types)...>,
        TAGSPEC(Types)...>;

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

1 Any entities declared or defined in namespace std in header <tuple> that are not already defined in namespace
   std::experimental::ranges_v1 in header <experimental/ranges++/tuple> are imported with
   using-declarations (7.3.3). [Example:
   namespace std { namespace experimental { namespace ranges_v1 {
      using std::tuple;
      using std::make_tuple;
      // ... others
   }}]
template <TaggedType... Types>
using tagged_tuple = tagged<
  tuple<TAGELEM(Types)...>,
  TAGSPEC(Types)...>;

Example:

// See 25.1:
tagged_tuple<tag::in(char*), tag::out(char*)> t{0, 0};
assert(&t.in() == &get<0>(t));
assert(&t.out() == &get<1>(t));

20.15.6.1 Tagged tuple creation functions

In the function descriptions that follow, let $i$ be in the range $[0, \text{sizeof...}(\text{Types})]$ in order and let $T_i$ be the $i^{th}$ type in a template parameter pack named $\text{Types}$, where indexing is zero-based.

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

2 Let $U_i$ be $\text{decay}_t<T_i>$ for each $T_i$ in Types. Then each $V_i$ in $\text{VTypes}$ is $X$& if $U_i$ equals $\text{reference}\_\text{wrapper}<X>$, otherwise $V_i$ is $U_i$.

3 Returns: tagged_tuple<Tags($\text{VTypes}$)...>(std::forward<Types>(t)...).

4 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&)>
24 Iterators library

24.1 General

This Clause describes components that C++ programs may use to perform iterations over containers (Clause 23), streams (27.7), and stream buffers (27.6), and ranges (24.10).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 3.

Table 3 — Iterators library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.2</td>
<td></td>
</tr>
<tr>
<td>24.7</td>
<td>&lt;experimental/ranges-v1/iterator&gt;</td>
</tr>
<tr>
<td>24.8</td>
<td></td>
</tr>
<tr>
<td>24.9</td>
<td></td>
</tr>
<tr>
<td>24.10</td>
<td></td>
</tr>
</tbody>
</table>

24.2.1 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 writable to the particular iterator type of \( i \). All iterators \( i \) for which the expression \( (*i).m \) is well-defined, support the expression \( i->m \) with the same semantics as \( (*i).m \). For every iterator type \( X \) for which equality is defined, there is a corresponding signed integer type called the difference type of the iterator.

Since iterators are an abstraction of pointers, their semantics is 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 International Standard defines five categories of iterators, according to the operations defined on them: weak input iterators, input iterators, weak output iterators, output iterators, forward iterators, bidirectional iterators and random access iterators, as shown in Table 4.

Table 4 — 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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3 The seven categories of iterators correspond to the iterator concepts WeakInputIterator, InputIterator, WeakOutputIterator, OutputIterator, ForwardIterator, BidirectionalIterator, and RandomAccessIterator, respectively. The generic term iterator refers to any type that satisfies WeakIterator.

4 ForwardInput iterators satisfy all the requirements of weak input iterators and can be used whenever a weak input iterator is specified; Forward iterators also 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.

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

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element
of the array, so for any iterator type there is an iterator value that points past the last element of a
corresponding sequence. These values are called past-the-end values. Values of an iterator \( i \) for which the
expression \(*i\) is defined are called dereferenceable. The library never assumes that past-the-end values are
dereferenceable. Iterators can also have singular values that are not associated with any sequence. [Example:
After the declaration of an uninitialized pointer \( x \) (as with int* \( x \)), \( x \) must always be assumed to have a
singular value of a pointer. —end example] Results of most expressions are undefined for singular values;
the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular
value to an iterator that holds a singular value, and for iterators that satisfy the DefaultConstructible
requirements. 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 sentinel is an abstraction of a past-the-end iterator. Sentinels are Regular types that can be used to
denote the end of a range. A sentinel and an iterator denoting a range shall be EqualityComparable. A
sentinel denotes an element when an iterator \( i \) compares equal to the sentinel, and \( i \) points to that element.

An iterator or sentinel \( j \) is called reachable from an iterator \( i \) if and only if there is a finite sequence of
applications of the expression \( ++i \) that makes \( i == j \). If \( j \) is reachable from \( i \), they refer to elements of the
same sequence.

Most of the library’s algorithmic templates that operate on data structures have interfaces that use ranges.
A range is a pair of iterators or an iterator and a sentinel that designate the beginning and end of the
computation. A range \([i,i)\) is an empty range; in general, a range \([i,j)\) 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
denoted by \( j \). Range \([i,j)\) is valid if and only if \( j \) is reachable from \( i \). 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). Therefore, requirement tables for the iterators do not have a complexity column.

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

An invalid iterator is an iterator that may be singular.2

In the following sections, \( a \) and \( b \) denote values of type \( X \) or const \( X \), difference_type and reference
refer to the types \(\text{iterator_traits}<X>::\text{difference_type}<X>\), \(\text{iterator_traits}<X>::\text{reference}<X>\), \(\text{difference_type_t}<X>\), and
\(\text{difference_type_t}<X>\), respectively, \( n \) denotes a value of
\(\text{difference_type}, u, \text{tmp}, \text{and } m \) denote identifiers, \( r \) denotes a value of \( Xk \), \( t \) denotes a value of value type \( T \), \( o \)
denotes a value of some type that is writable to the weak output iterator. [Note: For an iterator type \( X \) there
must be an instantiation of \( \text{iterator_traits}<X> \) (24.1.1) the type aliases \(\text{difference_type_t}<X>\)
and \(\text{difference_type_t}<X>\) must be well-formed. —end note]

### 24.2.2 Concept Readable

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

---

2) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been
invalidated is undefined.
template <class I>
concept bool Readable() {
    return Semiregular<I>() &
    requires (const I i) {
        typename ValueType value_type_t<I>;
        { i } -> const ValueType value_type_t<I>&; // pre: i is dereferenceable
    }
}

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

template <class> struct value_type { };  
template <class T>  
struct value_type<T*>  
    : enable_if<is_object<T>::value, remove_cv_t<T>> { };  
template<class I>  
struct value_type<I> : value_type<decay_t<I>> { };  
template <class I>  
struct value_type<I const> : value_type<decay_t<I>> { };  
template <class I>  
struct value_type<I volatile> : value_type<decay_t<I>> { };  
template <class I>  
struct value_type<I const volatile> : value_type<decay_t<I>> { };  
template <class T>  
requires requires { typename T::value_type; }  
struct value_type<T>  
    : enable_if<is_object<typename T::value_type>::value, typename T::value_type> { };  
template <class T>  
requires requires { typename T::element_type; }  
struct value_type<T>  
    : enable_if<is_object<typename T::element_type>::value, typename T::element_type> { };  

using ValueType value_type_t = typename value_type<T>::type;

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

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

When instantiated with a type I that has a nested type value_type such that T::value_type is valid and denotes a type, value_type<I>::type names that type, unless it is not an object type (3.9) in which case value_type<I> shall have no nested type type. [Note: Some legacy output iterators define a nested type named value_type that is an alias for void. These types are not Readable and have no associated value types. —end note]

When instantiated with a type I that has a nested type element_type, value_type<I>::type names that type, unless it is not an object type (3.9) in which case value_type<I> shall have no nested type type. [Note: Smart pointers like shared_ptr<int> are Readable and have an associated value type. But a smart pointer like shared_ptr<void> is not Readable and has no associated value type. —end note]

24.2.3 Concept MoveWritable [iterators.movewritable]

The MoveWritable concept describes the requirements for moving a value into an iterator’s referenced object.
concept bool MoveWritable() {
  return Semiregular<Out>() &&
  requires (Out o, T v) {
    *o = std::move(v); // not required to be equality preserving
  };
}

Let v be an rvalue or an lvalue of type (possibly const) T, and let o be a dereferenceable object of type Out. Then MoveWritable<Out, T>() is satisfied if and only if

(2.1) — If Out is Readable, then after the assignment *o = std::move(v), *o is equal to the value of v before the assignment.

After the expression *o = std::move(v), object o is not required to be dereferenceable.

v’s state is unspecified. [Note: v must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether v has been moved from or not. — end note]

24.2.4 Concept Writable

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

template <class Out, class T>
concept bool Writable() {
  return MoveWritable<Out, T>() &&
  requires (Out o, const T v) {
    *o = v; // not required to be equality preserving
  };
}

Let v be an lvalue of type (possibly const) T or an rvalue of type const T, and let o be a dereferenceable object of type Out. Then Writable<Out, T>() is satisfied if and only if

(2.1) — If Out is Readable, after the assignment *o = v, *o is equal to the value of v.

24.2.5 Concept IndirectlyMovable

The IndirectlyMovable concept describes the move relationship between a Readable type and a MoveWritable type.

template <class I, class Out>
concept bool IndirectlyMovable() {
  return Readable<I>() && MoveWritable<Out, ValueType<ValueType<T>>();
}

Let i be an object of type I, and let o be a dereferenceable object of type Out. Then IndirectlyMovable<I, Out>() is satisfied if and only if

(2.1) — If Out is Readable, after the assignment *o = std::move(*i), *o is equal to the value of *i before the assignment.

24.2.6 Concept IndirectlyCopyable

The IndirectlyCopyable concept describes the copy relationship between a Readable type and a Writable type.
template <class I, class Out>
concept bool IndirectlyCopyable() {
    return IndirectlyMovable<I, Out>() && Writable<Out, ValueType value_type_t<I>>();
}

Let \( i \) be an object of type \( I \), and let \( o \) be a dereferenceable object of type \( Out \). Then \( \text{IndirectlyCopyable}<I, Out>() \) is satisfied if and only if

\((2.1)\) — If \( Out \) is \( \text{Readable} \), after the assignment \( *o = *i \), \( *o \) is equal to the value of \( *i \).

### 24.2.7 Concept IndirectlySwappable [iterators.indirectlyswappable]

The \( \text{IndirectlySwappable} \) concept describes a swappable relationship between the reference types of two \( \text{Readable} \) types.

```cpp
template <class I1, class I2 = I1>
concept bool IndirectlySwappable() {
    return Readable<I1>() &&
    Readable<I2>() &&
    Swappable<ReferenceType reference_t<I1>, ReferenceType reference_t<I2>>();
}
```

### 24.2.8 Concept WeaklyIncrementable [iterators.weaklyincrementable]

The \( \text{WeaklyIncrementable} \) concept describes 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 \( \text{EqualityComparable} \).

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

Let \( i \) be an object of type \( I \). When both pre and post-increment are valid, \( i \) is said to be \( \text{incrementable} \). Then \( \text{WeaklyIncrementable}<I>() \) is satisfied if and only if

\((2.1)\) — ++\( i \) is valid if and only if \( i++ \) is valid.

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

\((2.3)\) — If \( i \) is incrementable, then \&++\( i \) == \&\( i \).

[Editor's note: Copied almost verbatim from the \text{InputIterator} description. This wording is removed there.]

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

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

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

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

1. If `bool(a == b)` then `bool(a++ == b)`.  
2. If `bool(a == b)` then `bool((a++, a) == ++b)`.  

[Editor’s note: Copied in part from the `ForwardIterator` description. This wording is removed there.]

24.2.10 Concept WeakIterator

The `WeakIterator` requirements concept forms the basis of the iterator concept taxonomy; every iterator satisfies the `WeakIterator` requirements. This set of requirements specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators (24.2.11), to read (24.2.14) or write (24.2.16) values, or to provide a richer set of iterator movements (24.2.17, 24.2.18, 24.2.19).

```
template <class I>
concept bool WeakIterator() {
  return WeaklyIncrementable<I>() &&
      requires(I i) {
        { *i } -> auto&&; // pre: i is dereferenceable
      };
}
```

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

24.2.11 Concept Iterator

The `Iterator` concept refines `WeakIterator` (24.2.10) and adds the requirement that the iterator is equality comparable.

In the `Iterator` concept, the set of values over which `==` is (required to be) defined can change over time. Each algorithm places additional requirements on the domain of `==` for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of `==` and `!=`. [Example: the call `find(a, b, x)` is defined only if the value of `a` has the property `p` defined as follows: `b` has property `p` and a value `i` has property `p` if (`*i==x`) or if (`*i!=x` and `++i` has property `p`). — end example]
template <class I>
concept bool Iterator() {
    return WeakIterator<I>() &&
    EqualityComparable<I>();
}

24.2.12 Concept Sentinel

The Sentinel concept defines requirements for a type that is an abstraction of the past-the-end iterator. Its values can be compared to an iterator for equality.

template <class T, class I>
concept bool Sentinel() {
    return Regular<T>() &&
    Iterator<I>() &&
    EqualityComparable<T, I>();
}

24.2.13 Concept WeakInputIterator

The WeakInputIterator concept is a refinement of WeakIterator (24.2.10). It defines requirements for a type whose referred to values can be read (from the requirement for Readable (24.2.2)) and which can be both pre- and post-incremented. However, weak input iterators are not required to be comparable for equality.

template <class I>
concept bool WeakInputIterator() {
    return WeakIterator<I>() &&
    Readable<I>() &&
    requires(I i, const I ci) {
        typename IteratorCategory<Iterator_category_t<I>>;
        requires DerivedFrom<IteratorCategory<Iterator_category_t<I>>, weak_input_iterator_tag>();
        { i++ } -> Readable; // not required to be equality preserving
        requires Same<ValueType,value_type_t<I>, ValueType,value_type_t<decltype(i++)>>();
        { *ci } -> const ValueType,value_type_t<I>&;
    };
}

24.2.14 Concept InputIterator

The InputIterator concept is a refinement of Iterator (24.2.11) and WeakInputIterator (24.2.13).

template <class I>
concept bool InputIterator() {
    return WeakInputIterator<I>() &&
    DerivedFrom<IteratorCategory<Iterator_category_t<I>>, input_iterator_tag>() &&
    Iterator<I>();
}

[Note: For input iterators, since InputIterator is only WeaklyIncrementable, \( a = b \) does not imply \( ++a = ++b \). (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value_type<T,ValueType,value_type_t<I>> is not required to be a CopyAssignable type (Table 19.4.7). These algorithms can be used with istreams as the source of the input data through the istream_iterator class template. — end note]
24.2.15 Concept WeakOutputIterator

The WeakOutputIterator concept is a refinement of WeakIterator (24.2.10). It defines requirements for a type that can be used to write values (from the requirement for Writable (24.2.4)) and which can be both pre- and post-incremented. However, weak output iterators are not required to satisfy EqualityComparable.

```
template <class I, class T>
concept bool WeakOutputIterator() {
    return WeakIterator<I>() && Writable<I, T>();
}
```

[Editor's note: The only valid use of an operator* is on the left side of the assignment statement. Assignment through the same value of the iterator happens only once. Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Equality and inequality might not be defined. Algorithms that take weak 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]

24.2.16 Concept OutputIterator

The OutputIterator concept is a refinement of Iterator (24.2.11) and WeakOutputIterator (24.2.15).

```
template <class I, class T>
concept bool OutputIterator() {
    return WeakOutputIterator<I, T>() && Iterator<I>();
}
```

[Editor's note: Output iterators are used by single-pass algorithms that write into a bounded range, like generate. — end note]

24.2.17 Concept ForwardIterator

A class or pointer type X satisfies the requirements of a forward iterator if

1. X satisfies the requirements of an input iterator (24.2.14),
2. X satisfies the DefaultConstructible requirements (17.6.3.1),
3. if X is a mutable iterator, reference is a reference to T; if X is a const iterator, reference is a reference to const T,
4. the expressions in Table 109 are valid and have the indicated semantics, and
5. objects of type X offer the multi-pass guarantee, described below.

The ForwardIterator concept refines InputIterator (24.2.14) and adds the multi-pass guarantee, described below.

```
template <class I>
concept bool ForwardIterator() {
    return InputIterator<I>() &&
        DerivedFrom<IteratorCategory<iterator_category_t<I>, forward_iterator_tag>() &
                    Incrementable<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]

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

1. `a == b` implies `++a == ++b` and
2. `X` is a pointer type or 

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

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

[Editor's note: Remove Table 109]

If `a` and `b` are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

If `a` and `b` are both dereferenceable, then `a == b` if and only if `*a` and `*b` are bound to the same object.

## 24.2.18 Concept BidirectionalIterator

A class or pointer type `X` satisfies the requirements of a bidirectional iterator if, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 110.

The BidirectionalIterator concept refines ForwardIterator (24.2.17), and adds the ability to move an iterator backward as well as forward.

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

[Editor's note: Remove table 110]

A bidirectional iterator `r` is decrementable if and only if there exists some `s` such that `++s == r`. The expressions `--r` and `r--` are only valid if `r` is decrementable.

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

1. `&--a == &a`.
2. If `bool(a == b)`, then `bool(a-- == b)`.
3. If `bool(a == b)`, then `bool((a--, a) == --b)`.
4. If `a` is incrementable and `bool(a == b)`, then `bool(--(++a) == b)`.
5. If `bool(a == b)`, then `bool(++(--a) == b)`.

[Note: Bidirectional iterators allow algorithms to move iterators backward as well as forward. — end note]
24.2.19 Concept RandomAccessIterator

A class or pointer type \( X \) satisfies the requirements of a random access iterator if, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 111.

The RandomAccessIterator concept refines BidirectionalIterator (24.2.18) 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.

```cpp
template <class I>
concept bool __MutableIterator = // exposition only
    Iterator<I>() &&
    requires(const I i) {
        *i -> auto&;
        i = *i;
    };

template <class I>
concept bool RandomAccessIterator() {
    return BidirectionalIterator<I>() &&
        DerivedFrom<IteratorCategory, iterator_category_t<I>, random_access_iterator_tag>() &&
        StrictTotallyOrdered<I>() &&
        SizedIteratorRange<I, I>() && (24.5.1)
        requires(I i, const I j, const DifferenceType<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>;
            i[n]; -> const ValueType<I>& Same<reference_t<I>>;
        } &&
        (!__MutableIterator<I> ||
         requires(const I i, const DifferenceType<I> n) { i[n] = *i; *i = i[n]; });
}
```

[Editor's note: Remove Table 111]

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

1. \((a += n)\) is equal to \( b \).
2. \((a += n)\) is equal to \&\( a \).
3. \((a + n)\) is equal to \((a += n)\).
4. For any two positive integers \( x \) and \( y \), if \( a + (x + y) \) is valid, then \( a + (x + y) \) is equal to \((a + x) + y \).
5. \( a + 0 \) is equal to \( a \).
6. \((a + (n - 1))\) is valid, then \( a + n \) is equal to ++\((a + (n - 1))\).
7. \((b += -n)\) is equal to \( a \).
8. \((b -= n)\) is equal to \( a \).
9. \&\((b -= n)\) is equal to \&\( b \).
— \((b - n)\) is equal to \(b -= n\).

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

24.3 Indirect callable requirements

24.3.1 In general

There are several concepts that group requirements of algorithms that take callable objects (20.9.2) as arguments.

[Editor's note: Specifying the algorithms in terms of these indirect callable concepts would ease the transition should we ever decide to support proxy iterators in the future. See the Future Work appendix (C.1).]

24.3.2 as_function_t type

The \(\text{FunctionType as\_function\_t}\) is an alias used to turn a callable type (20.9.1) into a function object type (20.9).

```cpp
// Exposition only
template <class T>
requires is_member_pointer<decay_t<T>>::value
auto __as_function(T&& t) {
    return mem_fn(t);
}
```

```cpp
template <class T>
T __as_function(T&& t) {
    return std::forward<T>(t);
}
```

```cpp
template <class T>
using FunctionType as\_function\_t = decltype(__as_function(declval<T>()));
```

24.3.3 Indirect callables

The indirect callable concepts are used to constrain those algorithms that accept callable objects (20.9.1) as arguments.

```cpp
template <class F, class...Is>
concept bool IndirectCallable() {
    return (Readable<Is>() && ...) &&
    Function<FunctionType as\_function\_t<F>, ValueType value\_type\_t<Is>...>();
}
```

```cpp
template <class F, class...Is>
concept bool IndirectRegularCallable() {
    return (Readable<Is>() && ...) &&
    RegularFunction<FunctionType as\_function\_t<F>, ValueType value\_type\_t<Is>...>();
}
```

```cpp
template <class F, class...Is>
concept bool IndirectCallablePredicate() {
    return (Readable<Is>() && ...) &&
    Predicate<FunctionType as\_function\_t<F>, ValueType value\_type\_t<Is>...>();
}
```

```cpp
template <class F, class I1, class I2 = I1>
```
concept bool IndirectCallableRelation() {
    return Readable<I1>() &&
    Readable<I2>() &&
    Relation<FunctionType as function_t<F>, ValueType value_type_t<I1>, ValueType value_type_t<I2>>()
}

template <class F, class I1, class I2 = I1>
concept bool IndirectCallableStrictWeakOrder() {
    return Readable<I1>() &&
    Readable<I2>() &&
    StrictWeakOrder<FunctionType as function_t<F>, ValueType value_type_t<I1>, ValueType value_type_t<I2>>()
}

template <class F, class...Is>
using IndirectCallableResultType =
    ResultType<FunctionType<F>, ValueType<Is>...>

class projected

24.3.4 Class template projected

The projected class template is intended for use when specifying the constraints of algorithms that accept
callable objects and projections. 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 ReferenceType of I.

template <Readable I, IndirectRegularCallable<I> Proj>
requires RegularFunction<FunctionType as function_t<Proj>, ValueType<reference_t<I>>>
structure projected {
    using value_type =
        decay_t<ResultType<FunctionType<Proj>, ValueType<I>>>;
    ResultType<FunctionType<Proj>, ReferenceType<reference_t<I>>> operator[](const);
    using value_type =
        decay_t<Result_of<as_function_t<F>(value_type_t<I>)>>>(reference_t<I>) const;
    result_of<as_function_t<F>(value_type_t<I>)>() operator*() const;
};

template <WeaklyIncrementable I, class Proj>
structure difference_type<Projected<I, Proj>> {
    using type = DifferenceType<difference_type_t<I>>;
};

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

24.4 Common algorithm requirements

24.4.1 In 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 four relational concepts for rearrange-
ments: Permutable, Mergeable, MergeMovableMoveMergeable, andSortable. There is one relational
concept for comparing values from different sequences: IndirectlyComparable.
[Note: The equal_to<> and less<> (20.9.6) function types used in the concepts below impose additional constraints on their arguments beyond those that appear explicitly in the concepts’ bodies. equal_to<> requires its arguments satisfy EqualityComparable (19.3.3), and less<> requires its arguments satisfy StrictTotallyOrdered (19.3.4). — end note]

24.4.2 Concept IndirectlyComparable [commonalgoreq.indirectlycomparable]

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

```cpp
template <class I1, class I2, class R = equal_to<>, class P1 = identity, class P2 = identity>
concept bool IndirectlyComparable() {
    return IndirectCallableRelation<R, projected<I1, P1>, projected<I2, P2>>();
}
```

24.4.3 Concept Permutable [commonalgoreq.permutable]

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

```cpp
template <class I>
concept bool Permutable() {
    return ForwardIterator<I>() && Movable<ValueType value_type_t<I>>() && IndirectlyMovable<I, I>();
}
```

24.4.4 Concept Mergeable [commonalgoreq.mergeable]

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

```cpp
template <class I1, class I2, class Out, class R = less<>, class P1 = identity, class P2 = identity>
concept bool Mergeable() {
    return InputIterator<I1>() && InputIterator<I2>() && WeaklyIncrementable<Out>() && IndirectlyCopyable<I1, Out>() && IndirectlyCopyable<I2, Out>() && IndirectlyCallableStrictWeakOrder<R, projected<I1, P1>, projected<I2, P2>>();
}
```

[Note: When less<> is used as the relation, the value type must satisfy TotallyOrdered. — end note]

24.4.5 Concept MoveMergeable [commonalgoreq.mergemovable]

The MoveMergeable concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by moving elements.

```cpp
template <class I1, class I2, class Out, class R = less<>, class P1 = identity, class P2 = identity>
concept bool MoveMergeable() {
    return InputIterator<I1>() && InputIterator<I2>() && WeaklyIncrementable<Out>() && IndirectlyMovable<I1, Out>() &&
```
IndirectlyMovable<II, Out>() &
IndirectCallableStrictWeakOrder<R, Pprojected<II, P1>, Pprojected<II, P2>>();
}

2  [Note: When less<> is used as the relation, the value type must satisfy TotallyOrdered. —end note]

24.4.6 Concept Sortable [commonalgoreq.sortable]
1 The Sortable concept specifies the common requirements of algorithms that permute sequences of iterators into ordered sequences (e.g., sort).

    template <class I, class R = less<>, class P = identity>
    concept bool Sortable() {
        return Permutable<I>() &&
        IndirectCallableStrictWeakOrder<R, Pprojected<I, P1>, Pprojected<I, P2>>();
    }

2  [Note: When less<> is used as the relation, the value type must satisfy TotallyOrdered. —end note]

24.5 Iterator range requirements [iteratorranges]
24.5.1 Concept SizedIteratorRange [iteratorranges.sizediteratorrange]
1 The SizedIteratorRange concept specifies the requirements on an Iterator (24.2.11) and a Sentinel that allows the use of the - operator to compute the distance between them in constant time.

    template <class I, class S>
    concept bool SizedIteratorRange() {
        return Sentinel<S, I>() &&
        requires (const I i, const S j) {  
            { i - i } -> DifferenceType difference_type_t<I>;
            { j - j } -> DifferenceType difference_type_t<I>;
            { i - j } -> DifferenceType difference_type_t<I>;
            { j - i } -> DifferenceType difference_type_t<I>;
        };
    }

2  Let a be a valid iterator of type I and b be a valid sentinel of type S. Let n be the smallest value of type DifferenceType<I> such that after n applications of ++a, then bool(a == b). Then SizedIteratorRange<I, S>() is satisfied if and only if:

(2.1) — (b - a) == n.
(2.2) — (a - b) == -n.
(2.3) — (a - a) == 0.
(2.4) — (b - b) == 0.

2 [Note: The SizedIteratorRange concept is satisfied by pairs of RandomAccessIterators (24.2.19) and by counted iterators and their sentinels (24.8.6.1). —end note]

[Editor’s note: This concept also gives us a way to demote the category of move_iterators to Input while retaining the ability of move_iterator pairs to communicate the range’s size to container constructors.]

24.6 Header <experimental/ranges/iterator> synopsis [iterator.synopsis]
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {

    // 24.7, primitives:
    template<class Iterator> struct using iterator_traits = see below;
    template<class T> struct iterator_traits<T>;

    template<class Category, class T, class Distance = ptrdiff_t,
    class Pointer = T*, class Reference = T&> struct iterator;

    template <class> struct difference_type;
    template <class> struct value_type;
    template <class> struct iterator_category;
    template <class WeaklyIncrementable> using DifferenceType
    = typename difference_type<WeaklyIncrementable>::type;
    template <class Readable> using ValueType
    = typename value_type<Readable>::type;
    template <class WeakInputIterator> using IteratorCategory
    = typename iterator_category<WeakInputIterator>::type;

    struct output_iterator_tag { };  
    struct weak_input_iterator_tag { }; 
    struct input_iterator_tag : public weak_input_iterator_tag { }; 
    struct forward_iterator_tag: public input_iterator_tag { }; 
    struct bidirectional_iterator_tag: public forward_iterator_tag { }; 
    struct random_access_iterator_tag: public bidirectional_iterator_tag { }; 

    // 24.7.5, iterator operations:
    template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n);
    template <class InputIterator>
    typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last);
    template <class ForwardIterator>
    ForwardIterator next(ForwardIterator x,
    typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
    template <class BidirectionalIterator>
    BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

    template <WeakIterator I>
    void advance(I& i, DifferenceType difference_type_t<I> n);
    template <Iterator I, Sentinel<I> S>
    void advance(I& i, S bound);
    template <Iterator I, Sentinel<I> S>
    DifferenceType difference_type_t<I> advance(I& i, DifferenceType difference_type_t<I> n, S bound);
    template <Iterator I, Sentinel<I> S>
    DifferenceType difference_type_t<I> distance(I first, S last);
    template <WeakIterator I>
    I next(I x, DifferenceType difference_type_t<I> n = 1);
    template <Iterator I, Sentinel<I> S>
    I next(I x, S bound);
    template <Iterator I, Sentinel<I> S>
    I next(I x, DifferenceType difference_type_t<I> n, S bound);
    template <BidirectionalIterator I>
    I prev(I x, DifferenceType difference_type_t<I> n = 1);
template <BidirectionalIterator I>
    I prev(I x, DifferenceType difference_t<I> n, I bound);

// 24.8, predefined iterators and sentinels:

// 24.8.1 Reverse iterators
template <class Iterator BidirectionalIterator I> class reverse_iterator;

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires EqualityComparable<I1, I2>()
    bool operator==(const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires EqualityComparable<I1, I2>()
    bool operator!=(const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires StrictTotallyOrdered<I1, I2>()
    bool operator<(const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires StrictTotallyOrdered<I1, I2>()
    bool operator<=(const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires SizedIteratorRange<I2, I1>()
    auto DifferenceType<I2> difference_t<I2> operator-(const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y) -> decltype(y.base() - x.base());

template <class Iterator RandomAccessIterator I>
    reverse_iterator<Iterator I> operator+(typename reverse_iterator<Iterator>::difference_type difference_t<I> n, const reverse_iterator<Iterator> & x);

template <class Iterator BidirectionalIterator I>
    reverse_iterator<Iterator I> make_reverse_iterator(Iterator I i);

// 24.8.2 Insert iterators
template <class Container> class back_insert_iterator;

§ 24.6
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, typename Container::iterator __IteratorType iterator _t<Container> i);

// 24.8.3 Move iterators
template <class IteratorWeakInputIterator I> class move_iterator;
template <class Iterator1InputIterator I1, class Iterator2InputIterator I2>
requires EqualityComparable<I1, I2>()
bool operator==(const move_iterator<Iterator1InputIterator I1>& x, const move_iterator<Iterator2InputIterator I2>& y);
template <class Iterator1InputIterator I1, class Iterator2InputIterator I2>
requires EqualityComparable<I1, I2>()
bool operator!=(const move_iterator<Iterator1InputIterator I1>& x, const move_iterator<Iterator2InputIterator I2>& y);
template <class Iterator1RandomAccessIterator I1, class Iterator2RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator<(const move_iterator<Iterator1RandomAccessIterator I1>& x, const move_iterator<Iterator2RandomAccessIterator I2>& y);
template <class Iterator1RandomAccessIterator I1, class Iterator2RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator<=(const move_iterator<Iterator1RandomAccessIterator I1>& x, const move_iterator<Iterator2RandomAccessIterator I2>& y);
template <class Iterator1RandomAccessIterator I1, class Iterator2RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator>(const move_iterator<Iterator1RandomAccessIterator I1>& x, const move_iterator<Iterator2RandomAccessIterator I2>& y);
template <class Iterator1RandomAccessIterator I1, class Iterator2RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator>=(const move_iterator<Iterator1RandomAccessIterator I1>& x, const move_iterator<Iterator2RandomAccessIterator I2>& y);

template <class Iterator1WeakInputIterator I1, class Iterator2WeakInputIterator I2>
requires SizedIteratorRange<I2, I1>()
auto difference_type_t<I1> difference_type_t<I2> operator-()
(const move_iterator<Iterator1InputIterator I1>& x, const move_iterator<Iterator2InputIterator I2>& y)->decltype(y.base() - x.base());
template <class Iterator1RandomAccessIterator I>
move_iterator<Iterator1>
operator+(typename move_iterator<Iterator1>::difference_type_difference_type_t<I1> difference_type_t<I1> n, const move_iterator<Iterator1>& x);
template <class Iterator1WeakInputIterator I>
move_iterator<Iterator1>
make_move_iterator<Iterator1 I> i);

// 24.8.4 Common iterators
template<class A, class B>
concept bool __WeaklyEqualityComparable = see below; // exposition only
template<class S, class I>
concept bool __WeakSentinel = see below; // exposition only

template <InputIterator I, __WeakSentinel<I> S> class common_iterator;

struct iterator_category<common_iterator<I, S>>;

template <InputIterator class I1, __WeakSentinel<I1> class S1, 
          InputIterator class I2, __WeakSentinel<I2> class S2>
requires EqualityComparable<I1, I2>() & __Weakly EqualityComparable<I1, S2> &
          __Weakly EqualityComparable<I2, S1>
bool operator==(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

template <InputIterator class I1, __WeakSentinel<I1> class S1, 
          InputIterator class I2, __WeakSentinel<I2> class S2>
requires EqualityComparable<I1, I2>() & __Weakly EqualityComparable<I1, S2> &
          __Weakly EqualityComparable<I2, S1>
bool operator!=(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

using __common_difference_type_t = see below; // exposition only

template <class I1, class S1, class I2, class S2>
concept bool __CommonSizedIteratorRange() { // exposition only
  return see below;
}

DifferenceType_t<I2> __common_difference_type_t<I1, I2> operator-(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

// 24.8.5 Default sentinels
class default_sentinel;
constexpr bool operator==(default_sentinel x, default_sentinel y) noexcept;
constexpr bool operator!=(default_sentinel x, default_sentinel y) noexcept;
constexpr bool operator<(default_sentinel x, default_sentinel y) noexcept;
constexpr bool operator<=(default_sentinel x, default_sentinel y) noexcept;
constexpr bool operator>(default_sentinel x, default_sentinel y) noexcept;
constexpr bool operator>=(default_sentinel x, default_sentinel y) noexcept;
constexpr ptrdiff_t operator-(default_sentinel x, default_sentinel y) noexcept;

// 24.8.6 Counted iterators

<WeakIterator I> class counted_iterator;

template <WeakIterator I> class counted_iterator;

template <WeakIterator I, WeakIterator I2>
requires Common<I1, I2>()
bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

template <WeakIterator I1, WeakIterator I2>
requires Common<I1, I2>()
bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);

template <WeakIterator I>
bool operator!=(
    const counted_iterator<I>& x, default_sentinel y);

template <WeakIterator I>
bool operator!=(
    default_sentinel x, const counted_iterator<I>& y);

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

template <WeakIterator I>
bool operator<(
    const counted_iterator<I>& x, default_sentinel y);

template <WeakIterator I>
bool operator<(
    default_sentinel x, const counted_iterator<I>& y);

template <WeakIterator I1, WeakIterator I2>
requires Common<I1, I2>()
bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

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

template <WeakIterator I>
bool operator>(
    const counted_iterator<I>& x, default_sentinel y);

template <WeakIterator I>
bool operator>(
    default_sentinel x, const counted_iterator<I>& y);

template <WeakIterator I1, WeakIterator I2>
requires Common<I1, I2>()
bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

template <WeakIterator I>
bool operator>(
    default_sentinel x, const counted_iterator<I>& y);

template <WeakIterator I, WeakIterator I2>
requires Common<I1, I2>()
DifferenceType difference_type_t<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

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

template <WeakIterator I>
counted_iterator<I> make_counted_iterator(I i, DifferenceType difference_type_t<I> n);

template <WeakIterator I>
void advance(counted_iterator<I>& i, DifferenceType difference_type_t<I> n);

// 24.8.8 Unreachable sentinels
struct unreachable { };

template <Iterator I>
constexpr bool operator==(const I&, unreachable) noexcept;

template <Iterator I>
constexpr bool operator!=(const I&, unreachable) noexcept;

template <Iterator I>
constexpr bool operator<(const I&, unreachable) noexcept;

template <Iterator I>
constexpr bool operator<=(const I&, unreachable) noexcept;

template <Iterator I>
constexpr bool operator>(const I&, unreachable) noexcept;

template <Iterator I>
constexpr bool operator>=(const I&, unreachable) noexcept;

§ 24.6
constexpr bool operator>=(unreachable, unreachable) noexcept;

// 24.8.7 Dangling wrapper
template <class T> class dangling;

// 24.9. stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
  class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
    const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(default_sentinel x,
    const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
    default_sentinel y);

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

§ 24.6
// 24.11, Range access:
template<class C> auto begin(C& c) -> decltype(c.begin());
template<class C> auto begin(const C& c) -> decltype(c.begin());
template<class C> auto end(C& c) -> decltype(c.end());
template<class C> auto end(const C& c) -> decltype(c.end());
template<T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
template<T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
using std::begin;
using std::end;
template<class>
concept bool _Auto = true; // exposition only

namespace {
  constexpr unspecified begin = unspecified;
  constexpr unspecified end = unspecified;
  constexpr unspecified cbegin = unspecified;
  constexpr unspecified cend = unspecified;
  constexpr unspecified rbegin = unspecified;
  constexpr unspecified rend = unspecified;
  constexpr unspecified crbegin = unspecified;
  constexpr unspecified crend = unspecified;
}

// 24.12, Range primitives:
namespace {
  constexpr unspecified size = unspecified;
  constexpr unspecified empty = unspecified;
  constexpr unspecified data = unspecified;
  constexpr unspecified cdata = unspecified;
}

namespace std {

§ 24.6 54

}}}
24.7 Iterator primitives [iterator.primitives]

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

24.7.1 Iterator traits associated types [iterator.assoc]

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 iterator_traits<Iterator>::difference_type, iterator_traits<Iterator>::value_type and iterator_traits<Iterator>::iterator_category be defined as the iterator’s difference type, value type and iterator category, respectively. In addition, the type

decltype(*declval<Readable&R&>())

shall be an alias for decltype((*declval<Readable&R&>())).

§ 24.7.1
shall be defined as the iterator's reference and pointer types, that is, for an iterator object \( a \), the same type as the type of \( *a \) and \( a-> \), respectively. In the case of an output iterator, the types

\[
\text{iterator_traits<Iterator>::difference_type}
\]

\[
\text{iterator_traits<Iterator>::value_type}
\]

\[
\text{iterator_traits<Iterator>::reference}
\]

\[
\text{iterator_traits<Iterator>::pointer}
\]

may be defined as void.

2 The template \texttt{iterator_traits<Iterator>} is defined as

```cpp
namespace std {
    template<class Iterator> struct iterator_traits {
        typedef typename Iterator::difference_type difference_type;
        typedef typename Iterator::value_type value_type;
        typedef typename Iterator::pointer pointer;
        typedef typename Iterator::reference reference;
        typedef typename Iterator::iterator_category iterator_category;
    };
}
```

3 It is specialized for pointers as

```cpp
namespace std {
    template<class T> struct iterator_traits<T*> {
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef T* pointer;
        typedef T& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
```

and for pointers to const as

```cpp
namespace std {
    template<class T> struct iterator_traits<const T*> {
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef const T* pointer;
        typedef const T& reference;
        typedef random_access_iterator_tag iterator_category;
    };
}
```

4 \texttt{DifferenceType<T> \_difference_type \_t<T>\_} is implemented as if:

```cpp
template <class> struct difference_type { };
template <class T>
struct difference_type<T*>
    : enable_if<is_object<T>::value, ptrdiff_t> { };
```

```cpp
template<>
struct difference_type<nullptr_t> {
    using type = ptrdiff_t;
};
```

```cpp
template<class I>
requires is_array<I>::value
```
struct difference_type<I> : difference_type<decay_t<I>> { }

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

template <class I>
struct difference_type<I volatile> : difference_type<decay_t<I>> { }

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

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

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

using DifferenceType_difference_type_t = typename difference_type<T>::type;

[Editor's note: REVIEW: The difference_type of unsigned Integral types is not large enough to cover the entire range. The Palo Alto report used a separate type trait for WeaklyIncrementable: DistanceType. DifferenceType_difference_type_t is only used for RandomAccessIterators. Cue discussion about the pros and cons of the two approaches.]

5 Users may specialize difference_type on user-defined types.

6 IteratorCategory_t<iterator_category_t<T>> is implemented as if:

template <class> struct iterator_category { };

struct iterator_category<T*> :
    enable_if<is_object<T>::value, random_access_iterator_tag> { }

struct iterator_category<T const> :
    iterator_category<T> { }

struct iterator_category<T volatile> :
    iterator_category<T> { }

struct iterator_category<T const volatile> :
    iterator_category<T> { }

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

using IteratorCategory_iterator_category_t = typename iterator_category<T>::type;

7 Users may specialize iterator_category on user-defined types.

8 If type T has a nested type iterator_category T::iterator_category is valid and denotes a type, then the type iterator_category<T>::type is computed as follows:

(8.1) If T::iterator_category is the same as or derives from std::random_access_iterator_tag, iterator_category<T>::type is ranges-v4::random_access_iterator_tag.
(8.2) — Otherwise, if \(T::\text{iterator\_category}\) is the same as or derives from \(\text{std::bidirectional\_iterator\_tag}\), \(\text{iterator\_category}\<T>::\text{type}\) is \(\text{ranges\_v1::bidirectional\_iterator\_tag}\).

(8.3) — Otherwise, if \(T::\text{iterator\_category}\) is the same as or derives from \(\text{std::forward\_iterator\_tag}\), \(\text{iterator\_category}\<T>::\text{type}\) is \(\text{ranges\_v1::forward\_iterator\_tag}\).

(8.4) — Otherwise, if \(T::\text{iterator\_category}\) is the same as or derives from \(\text{std::input\_iterator\_tag}\), \(\text{iterator\_category}\<T>::\text{type}\) is \(\text{ranges\_v1::input\_iterator\_tag}\).

(8.5) — Otherwise, if \(T::\text{iterator\_category}\) is the same as or derives from \(\text{std::output\_iterator\_tag}\), \(\text{iterator\_category}\<T>\) has no nested type.

(8.6) — Otherwise, \(\text{iterator\_category}\<T>::\text{type}\) is \(T::\text{iterator\_category}\).

[Note: If there is an additional pointer type \_\_\_far such that the difference of two \_\_\_far is of type \textit{long}, an implementation may define]

```cpp
template<class T> struct iterator_traits<T \_\_\_far*> {
    typedef long difference\_type;
    typedef T value\_type;
    typedef T \_\_\_far* pointer;
    typedef T \_\_\_far& reference;
    typedef random\_access\_iterator\_tag iterator\_category;
};
```

— end note]}

10 For the sake of backwards compatibility, this standard document specifies the existence of an \textit{iterator\_traits} alias that collects an iterator’s associated types. It is defined as if:

```cpp
template <WeakInputIterator I> struct __pointer\_type {
    using type = add_pointer\_t<ReferenceType reference\_t<I>>;
};

template <WeakInputIterator I>
requires requires(I i) { { i.operator->() } -> auto&&; }
struct __pointer\_type<I> {
    using type = decltype(declval<I>().operator->());
};

template <class> struct __iterator\_traits { };
// exposition only

template <WeakInputIterator I> struct __iterator\_traits<I> {
    using difference\_type = DifferenceType difference\_type\_t<I>;
    using value\_type = void;
    using reference = void;
    using pointer = void;
    using iterator\_category = output\_iterator\_tag;
};

// exposition only

template <WeakInputIterator I> struct __iterator\_traits<I> {
    using difference\_type = DifferenceType difference\_type\_t<I>;
    using value\_type = ValueType value\_type\_t<I>;
    using reference = ReferenceType reference\_t<I>;
    using pointer = typename __pointer\_type<I>::type;
    using iterator\_category = IteratorCategory iterator\_category\_t<I>;
};

template <class I>
using iterator\_traits = __iterator\_traits<I>;
```
11 [Note: `iterator_traits` is an alias template to intentionally break code that tries to specialize it to prevent user code from specializing it. — end note]

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

```
template <class BidirectionalIterator I>
void reverse(BidirectionalIterator I first, BidirectionalIterator I last) {
    typename iterator_traits<BidirectionalIterator>::difference_type I_difference_type_t<I> n =
    distance(first, last);
    --n;
    while(n > 0) {
        typename iterator_traits<BidirectionalIterator>::value_type ValueType<I> tmp = *first;
        *first++ = *--last;
        *last = tmp;
        n -= 2;
    }
}
```

—end example]

24.7.2 Standard iterator traits

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

namespace std {
    template <experimental::ranges_v1::WeakIterator Out>
    struct iterator_traits<Out> {
        using difference_type = experimental::ranges_v1::DifferenceType<Out>;
        using value_type = see below;
        using reference = see below;
        using pointer = see below;
        using iterator_category = std::output_iterator_tag;
    };
}

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

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

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

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

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

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

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

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

§ 24.7.2
Otherwise, std::iterator_traits<Out>::pointer is void.

```cpp
template <experimental::ranges_v1::WeakInputIterator WeakIn>
struct iterator_traits<WeakIn> { }
```

```cpp
template <experimental::ranges_v1::InputIterator In>
struct iterator_traits<In> {
  using difference_type = experimental::ranges_v1::DifferenceType_difference_type_t<In>;
  using value_type = experimental::ranges_v1::ValueType_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:

- If type In has a nested type reference In::reference is valid and denotes a type, then std::iterator_traits<In>::reference is In::reference.
- Otherwise, std::iterator_traits<In>::reference is experimental::ranges_v1::ReferenceType_reference_t<In>.

The nested type pointer is computed as follows:

- If type In has a nested type pointer In::pointer is valid and denotes a type, then std::iterator_traits<In>::pointer is In::pointer.
- Otherwise, std::iterator_traits<In>::pointer is experimental::ranges_v1::iterator_traits<In>::pointer.

Let type C be experimental::ranges_v1::IteratorCategory_iterator_category_t<In>. The nested type std::iterator_traits<In>::iterator_category is computed as follows:

- 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.
- Otherwise, if experimental::ranges_v1::ReferenceType_reference_t<In> is not a reference type, std::iterator_traits<In>::iterator_category is std::input_iterator_tag.
- Otherwise, if C is the same as or inherits from experimental::ranges_v1::random_access_iterator_tag, std::iterator_traits<In>::iterator_category is std::random_access_iterator_tag.
- Otherwise, if C is the same as or inherits from experimental::ranges_v1::bidirectional_iterator_tag, std::iterator_traits<In>::iterator_category is std::bidirectional_iterator_tag.
- Otherwise, if C is the same as or inherits from experimental::ranges_v1::forward_iterator_tag, std::iterator_traits<In>::iterator_category is std::forward_iterator_tag.
- 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. — end note]
24.7.3 Basic iterator

The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```cpp
namespace std {
    template<class Category, class T, class Distance = ptrdiff_t,
       class Pointer = T*, class Reference = T&>
    struct iterator {
        typedef T value_type;
        typedef Distance difference_type;
        typedef Pointer pointer;
        typedef Reference reference;
        typedef Category iterator_category;
    }
}
```

24.7.4 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: `weak_input_iterator_tag`, `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every weak input iterator of type `Iterator`, `iterator_traits<Iterator>::iterator_category` shall be defined to be the most specific category tag that describes the iterator’s behavior.

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
    struct output_iterator_tag { }; struct weak_input_iterator_tag { }; struct input_iterator_tag : public weak_input_iterator_tag { }; struct forward_iterator_tag : public input_iterator_tag { }; struct bidirectional_iterator_tag : public forward_iterator_tag { }; struct random_access_iterator_tag : public 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 `iterator_traits` `difference_type`, `value_type`, and `iterator_category` templates:

```cpp
template<class T> struct iterator_traits<BinaryTreeIterator<T>> { 
    typedef std::ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef bidirectional_iterator_tag iterator_category;
};
```

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

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

Typically, however, it would be easier to derive BinaryTreeIterator<T> from iterator<bidirectional_iterator_tag, T, ptdiff_t, T*, T&>.

Example: If evolve() is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is as follows:

```
template <class BidirectionalIterator>
inline void evolve(BidirectionalIterator first, BidirectionalIterator last) {
   evolve(first, last,
       typename iterator_traits<BidirectionalIterator>::iterator_category());
}
```

```
template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last, bidirectional_iterator_tag) {
   // more generic, but less efficient algorithm
}
```

```
template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last, random_access_iterator_tag) {
   // more efficient, but less generic algorithm
}
```

Example: If a C++ program wants to define a bidirectional iterator for some data structure containing double and such that it works on a large memory model of the implementation, it can do so with:

```
class MyIterator :
   public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
      // code implementing ++, etc.
   };
```

Then there is no need to specialize the iterator_traits template.

24.7.5 Iterator operations

Since only random access iterator types that satisfy RandomAccessIterator provide the + and operator, and types that satisfy SizedIteratorRange provide the - operators, the library provides four function templates advance, and distance, next, and prev. These function templates use + and - for random access iterators and sized iterator ranges, respectively (and are, therefore, constant time for them); for weak output, weak input, input, forward and bidirectional iterators they use ++ to provide linear time implementations.

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
```

```
template <WeakIterator I>
void advance(I& i, DifferenceType difference_type_t<I> n);
```

Requires: n shall be negative only for bidirectional and random access iterators.

Effects: Increments (or decrements for negative n) iterator reference i by n.
template <Iterator I, Sentinel<I> S>
void advance(I& i, S bound);

4 Requires: bound shall be reachable from i or i shall be reachable from bound and is_assignable<I &, S &&>::value is true.

5 Effects:

(5.1) — If I and S are the same type, this function is constant time is_assignable<I &, S &&>::value is true this function is equivalent to i = std::move(bound).

(5.2) — If otherwise, if SizedIteratorRange<I, S>() is satisfied, this function shall dispatch to is equivalent to advance(I, bound - i).

(5.3) — Otherwise, this function increments iterator reference i until i == bound.

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

6 Requires: n shall be negative only for bidirectional and random access iterators. If n is negative, i shall be reachable from bound; otherwise, bound shall be reachable from i.

7 Effects:

(7.1) — If SizedIteratorRange<I, S>() is satisfied:

(7.1.1) — If (0 <= n ? n >= D : n <= D) is true, where D is bound - i, this function dispatches is equivalent to advance(i, bound),

(7.1.2) — Otherwise, this function dispatches is equivalent to advance(i, n).

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

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

template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type distance(InputIterator first, InputIterator last);

9 template <Iterator I, Sentinel<I> S>
DifferenceType difference_type_t<I> distance(I first, S last);

10 Effects: If InputIterator meets the requirements of random access iterator SizedIteratorRange<I, S>() is satisfied, returns (last - first); otherwise, returns the number of increments needed to get from first to last.

11 Requires: If InputIterator meets the requirements of random access iterator SizedIteratorRange<I, S>() is satisfied last shall be reachable from first or first shall be reachable from last; otherwise, last shall be reachable from first.

template <class ForwardIterator>
ForwardIterator next(ForwardIterator x, typename std::iterator_traits<ForwardIterator>::difference_type n = 1);

12 template <WeakIterator I>
I next(I x, DifferenceType difference_type_t<I> n = 1);

13 Effects: Equivalent to advance(x, n); return x;

template <Iterator I, Sentinel<I> S>
I next(I x, S bound);
12  \textit{Effects:} Equivalent to $\text{advance}(x, \text{bound}); \text{return } x;\$

\begin{verbatim}
  template <Iterator I, Sentinel<I> S>
  I next(I x, DifferenceType difference_type_t<I> n, S bound);
\end{verbatim}

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

\begin{verbatim}
  template <class BidirectionalIterator>
  BidirectionalIterator prev(BidirectionalIterator x,
  typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
\end{verbatim}

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

\begin{verbatim}
  template <BidirectionalIterator I>
  I prev(I x, DifferenceType difference_type_t<I> n = 1);
\end{verbatim}

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

\section{24.8 Iterator adaptors \[iterators.predef\]}

\subsection{24.8.1 Reverse iterators \[iterators.reverse\]}

Class template \texttt{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 iterator $i$ is established by the identity: \texttt{&*(\texttt{reverse}\_\texttt{iterator}(i))} == \texttt{&*(i - 1)}.

\subsection{24.8.1.1 Class template \texttt{reverse\_iterator} \[reverse.iterator\]}

\begin{verbatim}
namespace std {
  namespace experimental {
    namespace ranges_v1 {
      inline namespace v1 {
        template <class Iterator BidirectionalIterator I>
        class reverse_iterator {
        public:

          typedef Iterator iterator_type;
          typedef typename iterator_traits<Iterator>::difference_type difference_type;
          typedef typename iterator_traits<Iterator>::reference reference;
          typedef typename iterator_traits<Iterator>::pointer pointer;

          using iterator_type = I;
          using difference_type = DifferenceType difference_type_t<I>;
          using value_type = ValueType value_type_t<I>;
          using iterator_category = IteratorCategory iterator_category_t<I>;
          using reference = ReferenceType reference_t<I>;
          using pointer = I;
\end{verbatim}
reverse_iterator();
explicit reverse_iterator(Iterator I x);
template <class BidirectionalIterator U>
    requires ConvertibleTo<U, I>()
reverse_iterator(const reverse_iterator<U>& u);
template <class BidirectionalIterator U>
    requires ConvertibleTo<U, I>()
reverse_iterator& operator=(const reverse_iterator<U>& u);

Iterator I base() const; // explicit
reference operator*() const;
pointer operator->() const;

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

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

protected
private:
    Iterator I current; // exposition only
}

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires EqualityComparable<I1, I2>()
bool operator==(const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
    requires StrictTotallyOrdered<I1, I2>()
bool operator<(const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
    requires EqualityComparable<I1, I2>()
bool operator!=(const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
    requires StrictTotallyOrdered<I1, I2>()
bool operator>(const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y);

template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
    requires StrictTotallyOrdered<I1, I2>()
bool operator>=(const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y);
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1>RandomAccessIterator I1, class Iterator2>RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator<=(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1BidirectionalIterator I1, class Iterator2BidirectionalIterator I2>
requires SizedIteratorRange<I2, I1>()
autoDifferenceTypeIDifference_type_t<I2> operator-(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y)--x.base()-y.base();
template <class Iterator>RandomAccessIterator I>
reverse_iterator<Iterator>
operator+( typename reverse_iterator<Iterator>::difference_type_difference_type_t<Iterator>_difference_type_t<I> n,
const reverse_iterator<Iterator>& x);
template <class Iterator>BidirectionalIterator I>
reverse_iterator<Iterator> make_reverse_iterator(I i);
}
}}

24.8.1.2 reverse_iterator requirements [reverse.iter.requirements]

1 The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (24.2.18).

2 Additionally, Iterator shall meet the requirements of a Random Access Iterator (24.2.19) if any of the members operator+ (24.8.1.3.8), operator- (24.8.1.3.10), operator+= (24.8.1.3.9), operator-= (24.8.1.3.11), operator[] (24.8.1.3.12), or the global operators operator< (24.8.1.3.14), operator> (24.8.1.3.16), operator<= (24.8.1.3.18), operator>= (24.8.1.3.17), operator- (24.8.1.3.19) or operator+ (24.8.1.3.20) are referenced in a way that requires instantiation (14.7.1).

24.8.1.3 reverse_iterator operations [reverse.iter.ops]

24.8.1.3.1 reverse_iterator constructor [reverse.iter.cons]

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 Iterator.

explicit reverse_iterator(Iterator x);

2 Effects: Initializes current with x.

template <class BidirectionalIterator U>
requiresConvertibleTo<U, I>()
reverse_iterator(const reverse_iterator<U>& u);

3 Effects: Initializes current with u.current.

24.8.1.3.2 reverse_iterator::operator=[reverse.iter.op=]

template <class BidirectionalIterator U>
requiresConvertibleTo<U, I>()
reverse_iterator&
operator=(const reverse_iterator<U>& u);

§ 24.8.1.3.2
Effects: Assigns \texttt{u.base()} \texttt{current} to \texttt{current}.

Returns: \texttt{*this}.

24.8.1.3.3 Conversion [reverse.iter.conv]

\begin{verbatim}
iterator
base() const;  // explicit
\end{verbatim}

Returns: \texttt{current}.

24.8.1.3.4 operator* [reverse.iter.op.star]

reference operator*() const;

Effects: Equivalent to \texttt{*prev(current)}

\begin{verbatim}
iterator tmp = current;
return -*tmp;
\end{verbatim}

24.8.1.3.5 operator-> [reverse.iter.opref]

pointer operator->() const;

Returns: Effects: Equivalent to \texttt{std::addressof(operator*)} \texttt{prev(current)}.

24.8.1.3.6 operator++ [reverse.iter.op++]

reverse_iterator& operator++();

Effects: \texttt{--current};

Returns: \texttt{*this}.

reverse_iterator operator++(int);

Effects:

\begin{verbatim}
reverse_iterator tmp = *this;
--current;
return tmp;
\end{verbatim}

24.8.1.3.7 operator-- [reverse.iter.op--]

reverse_iterator& operator--();

Effects: \texttt{++current}

Returns: \texttt{*this}.

reverse_iterator operator--(int);

Effects:

\begin{verbatim}
reverse_iterator tmp = *this;
++current;
return tmp;
\end{verbatim}

24.8.1.3.8 operator+ [reverse.iter.op+]

reverse_iterator
operator+(typename reverse_iterator<Iterator>::difference_type n) const+

requires RandomAccessIterator<Iterator>();

Returns: \texttt{reverse_iterator(current-n)}.

§ 24.8.1.3.8
24.8.1.3.9  operator+=
reverse_iterator&
operator+=(typename reverse_iterator<Iterator>::difference_type n) +
requires RandomAccessIterator<Iterator>();
1  Effects: current -= n;
2  Returns: *this.

24.8.1.3.10  operator-
reverse_iterator
operator-(typename reverse_iterator<Iterator>::difference_type n) const +
requires RandomAccessIterator<Iterator>();
1  Returns: reverse_iterator(current+n).

24.8.1.3.11  operator-=
reverse_iterator&
operator-=(typename reverse_iterator<Iterator>::difference_type n) +
requires RandomAccessIterator<Iterator>();
1  Effects: current += n;
2  Returns: *this.

24.8.1.3.12  operator[]
unspecified reference operator[](typename reverse_iterator<Iterator>::difference_type n) const +
requires RandomAccessIterator<Iterator>();
1  Returns: current[-n-1].

24.8.1.3.13  operator==
template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
requires EqualityComparable<I1, I2>();
bool operator==(const reverse_iterator<Iterator1>I1&, const reverse_iterator<Iterator2>I2&) y);
1  Returns: Effects: Equivalent to x.current == y.current.

24.8.1.3.14  operator<
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>();
bool operator<(const reverse_iterator<Iterator1>I1&, const reverse_iterator<Iterator2>I2&) y);
1  Returns: Effects: Equivalent to x.current > y.current.

24.8.1.3.15  operator!=
template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
requires EqualityComparable<I1, I2>();
bool operator!=(const reverse_iterator<Iterator1>I1&, const reverse_iterator<Iterator2>I2&) y);
24.8.1.3.16 operator>

```
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator>(
    const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y);
```

**Returns:** Equivalent to \( x.current != y.current \).

**Effects:**

Equivalent to \( x.current < y.current \).

24.8.1.3.17 operator>=

```
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator>= (const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y);
```

**Returns:** Equivalent to \( x.current \leq y.current \).

24.8.1.3.18 operator<=

```
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator<= (const reverse_iterator<Iterator1 I1>& x, const reverse_iterator<Iterator2 I2>& y);
```

**Returns:** Equivalent to \( x.current \geq y.current \).

24.8.1.3.19 operator-

```
template <class Iterator1 BidirectionalIterator I1, class Iterator2 BidirectionalIterator I2>
requires SizedIteratorRange<I2, I1>()
auto DifferenceType<I2> difference_type_t<I2> operator-(
    const reverse_iterator<Iterator1 I1>& x,
    const reverse_iterator<Iterator2 I2>& y)
->decltype(y.base() - x.base());
```

**Returns:** Equivalent to \( y.current - x.current \).

24.8.1.3.20 operator+

```
template <class Iterator1 RandomAccessIterator I>
reverse_iterator<Iterator1> operator+(typename reverse_iterator<Iterator1>::difference_type_t<I> difference_type_t<I> n, const reverse_iterator<Iterator1>& x);
```

**Returns:** Equivalent to \( \text{reverse_iterator<Iterator1>}(x.current - n) \).

24.8.1.3.21 Non-member function make_reverse_iterator()

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

**Returns:** \( \text{reverse_iterator<Iterator>}(i) \).
24.8.2 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,

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

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

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 the requirements of output iterator `WeakOutputIterator`. `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.

### 24.8.2.1 Class template back_insert_iterator

[back.insert.iterator]

[Editor's note: REVIEW: Re-specify this in terms of a Container concept? Or Range? Or leave it?]

```c
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
  template <class Container>
  class back_insert_iterator {
    public:
      iterator<output_iterator_tag, void, void, void, void> _;
    protected:
      Container* container; // exposition only
    public:
      typedef Container using container_type = Container;
      using difference_type = ptrdiff_t;
      using iterator_category = output_iterator_tag;
      constexpr back_insert_iterator();
      explicit back_insert_iterator(Container& x);
      back_insert_iterator<Container>& operator=(const typename Container::value_type& value);
      back_insert_iterator<Container>& operator=(typename Container::value_type&& value);
      back_insert_iterator<Container>& operator=(typename Container::value_type& value);
      back_insert_iterator<Container>& operator*();
      back_insert_iterator<Container>& operator++();
      back_insert_iterator<Container>& operator++(int);
    };

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

### 24.8.2.2 back_insert_iterator operations

[back.insert.iter.ops]

### 24.8.2.2.1 back_insert_iterator constructor

[back.insert.iter.cons]

```c
constexpr back_insert_iterator();
```
Effects: Value-initializes container.

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

24.8.2.2.2 back_insert_iterator::operator=
    [back.insert.iter.op=]

    back_insert_iterator<Container>&
    operator=(const typename Container::value_type& value);
    Effects: Equivalent to container->push_back(value);
    Returns: *this.

    back_insert_iterator<Container>&
    operator=(typename Container::value_type&& value);
    Effects: Equivalent to container->push_back(std::move(value));
    Returns: *this.

24.8.2.2.3 back_insert_iterator::operator*
    [back.insert.iter.op*]

    back_insert_iterator<Container>& operator*();
    Returns: *this.

24.8.2.2.4 back_insert_iterator::operator++
    [back.insert.iter.op++]

    back_insert_iterator<Container>& operator++();
    back_insert_iterator<Container> operator++(int);
    Returns: *this.

24.8.2.2.5 back_inserter
    [back.inserter]

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

24.8.2.3 Class template front_insert_iterator
    [front.insert.iterator]

    namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
    template <class Container>
    class front_insert_iterator +{
        public: iterator<output_iterator_tag, void, void, void, void>;
        protected:
            Container* container; // exposition only
           
            typedef Container using container_type = Container;
            using difference_type = ptrdiff_t;
            using iterator_category = output_iterator_tag;
        constexpr front_insert_iterator();
        explicit front_insert_iterator(Container& x);
        front_insert_iterator<Container>&
            operator=(const typename Container::value_type& value);
        front_insert_iterator<Container>&
            operator=(typename Container::value_type&& value);
    }}}}
front_insert_iterator<Container>& operator*();
front_insert_iterator<Container>& operator++();
front_insert_iterator<Container> operator++(int);
}

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

24.8.2.4  front_insert_iterator operations  [front.insert.iter.ops]

24.8.2.4.1  front_insert_iterator constructor  [front.insert.iter.cons]

constexpr front_insert_iterator();

Effects: Value-initializes container.

explicit front_insert_iterator(Container& x);  

Effects: Initializes container with std::addressof(x).

24.8.2.4.2  front_insert_iterator::operator=

front_insert_iterator<Container>&
operator=(const typename Container::value_type& value);

Effects: Equivalent to container->push_front(value);

1 Returns: *this.

front_insert_iterator<Container>&
operator=(typename Container::value_type&& value);

Effects: Equivalent to container->push_front(std::move(value));

4 Returns: *this.

24.8.2.4.3  front_insert_iterator::operator*  [front.insert.iter.op=*]

front_insert_iterator<Container>& operator*();

Returns: *this.

24.8.2.4.4  front_insert_iterator::operator++  [front.insert.iter.op++]

front_insert_iterator<Container>& operator++();
front_insert_iterator<Container> operator++(int);

Returns: *this.

24.8.2.4.5  front_inserter  [front inserter]

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

Returns: front_insert_iterator<Container>(x).

24.8.2.5  Class template insert_iterator  [insert.iterator]

namespace std {
    namespace experimental {
        namespace ranges_v1 {
            inline namespace v1 {
                template <class Container>
                class insert_iterator {
                    public_iterator<output_iterator_tag, void, void, void, void>
                }
            }
        }
    }
}
protected private:
    Container* container;  // exposition only
    typename Container::iterator iter;  // exposition only

public:
    typedef Container using container_type = Container;
    using difference_type = ptdiff_t;
    using iterator_category = output_iterator_tag;
    insert_iterator();
    insert_iterator(Container& x, typename Container::iterator i);
    insert_iterator<Container>&
        operator=(const typename Container::value_type& value);
    insert_iterator<Container>&
        operator=(typename Container::value_type&& value);
    insert_iterator<Container>& operator*();
    insert_iterator<Container>& operator++();
    insert_iterator<Container>& operator++(int);
};

template <class Container>
    insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
}
}

24.8.2.6 insert_iterator operations [insert.iter.ops]
24.8.2.6.1 insert_iterator constructor [insert.iter.cons]
insert_iterator();
1  Effects: Value-initializes container and iter.

insert_iterator(Container& x, typename Container::iterator i);
2  Requires: i is an iterator into x.
3  Effects: Initializes container with std::addressof(x) and iter with i.

24.8.2.6.2 insert_iterator::operator=
1  Equivalent to.

insert_iterator<Container>&
    operator=(const typename Container::value_type& value);
1  Effects: Equivalent to.

    iter = container->insert(iter, value);
    ++iter;
2  Returns: *this.

insert_iterator<Container>&
    operator=(typename Container::value_type&& value);
3  Equivalent to.

    iter = container->insert(iter, std::move(value));
    ++iter;
4  Returns: *this.
24.8.2.6.3 insert_iterator::operator*

insert_iterator<Container>& operator*();

Returns: *this.

24.8.2.6.4 insert_iterator::operator++

insert_iterator<Container>& operator++();
insert_iterator<Container>& operator++(int);

Returns: *this.

24.8.2.6.5 inserter

template <class Container>
insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

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

24.8.3 Move iterators

Class template move_iterator is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator’s indirection operator to an value reference of the value type. Some generic algorithms can be called with move iterators to replace copying with moving. [Editor's note: This is untrue now. The algorithms that do copying are constrained with IndirectlyCopyable, which will reject move iterators.]

Example:

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

— end example]

24.8.3.1 Class template move_iterator

namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {

template <class Iterator, WeakInputIterator I>
requires Same<ReferenceType<I>, ValueType<I>&>()
class move_iterator {
public:

    typedef Iterator iterator_type;
    typedef typename iterator_traits<Iterator>::difference_type difference_type;
    typedef Iterator pointer;
    typedef typename iterator_traits<Iterator>::value_type value_type;
    typedef typename iterator_traits<Iterator>::iterator_category iterator_category;

    typedef value_type&& reference;

    using iterator_type = I;
    using difference_type = DifferenceType<difference_type_t<I>>;
    using value_type = ValueType<value_type_t<I>>;
    using iterator_category = IteratorCategory<I>& see below;
    using reference = ValueType<>& see below;
    using pointer = I;

§ 24.8.3.1
move_iterator();
explicit move_iterator(Iterator I i);
template <class WeakInputIterator U>
  requires ConvertibleTo<Iterator, U>()
move_iterator(const move_iterator<U>& u);
template <class WeakInputIterator U>
  requires ConvertibleTo<Iterator, U>()
move_iterator& operator=(const move_iterator<U>& u);

iterator_base() const;
reference operator*() const;
pointer operator->() const;

move_iterator& operator++();
move_iterator operator++(int);
move_iterator& operator--();
  requires BidirectionalIterator<Iterator>();
move_iterator operator--(int);
  requires BidirectionalIterator<Iterator>();

move_iterator operator+(difference_type n) const;
  requires RandomAccessIterator<Iterator>();
move_iterator& operator+=(difference_type n);
  requires RandomAccessIterator<Iterator>();
move_iterator operator-(difference_type n) const;
  requires RandomAccessIterator<Iterator>();
move_iterator& operator-=(difference_type n);
  requires RandomAccessIterator<Iterator>();
unspecified_reference operator[](difference_type n) const;
  requires RandomAccessIterator<Iterator>();

private:
  iterator I current;  // exposition only
};

template <class Iterator1 InputIterator I1, class Iterator2 InputIterator I2>
  requires EqualityComparable<I1, I2>()
bool operator==(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);
template <class Iterator1 InputIterator I1, class Iterator2 InputIterator I2>
  requires EqualityComparable<I1, I2>()
bool operator!=(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
  requires StrictTotallyOrdered<I1, I2>()
bool operator<(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
  requires StrictTotallyOrdered<I1, I2>()
bool operator<=(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
  requires StrictTotallyOrdered<I1, I2>()
bool operator>(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
  requires StrictTotallyOrdered<I1, I2>()
bool operator>(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);
### move_iterator requirements

1. If `is_base_of<input_iterator_tag, iterator_category_t<I>>::value` is true, the template specialization `move_iterator<I>` shall define the nested type named `iterator_category` as a synonym for `input_iterator_tag`, otherwise as a synonym for `weak_input_iterator_tag`.

2. Let `R` be `reference_t<I>`. If `is_reference<R>::value` is true, the template specialization `move_iterator<I>` shall define the nested type named `reference` as a synonym for `remove_reference_t<R>&&`, otherwise as a synonym for `R`.

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]

### move_iterator operations

#### move_iterator constructors

1. **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 `Iterator`.

2. **explicit move_iterator(Iterator I i);**
   
   **Effects:** Constructs a `move_iterator`, initializing `current` with `i`.

3. **template <class WeakInputIterator I>
   requiresConvertibleTo<I, I>()
   move_iterator(const move_iterator<I>& u);**
   
   **Effects:** Constructs a `move_iterator`, initializing current with `u.base()`.`u.current`.

4. **Requires:** `U` shall be convertible to `Iterator`.

---

1. If `is_base_of<input_iterator_tag, iterator_category_t<I>>::value` is true, the template specialization `move_iterator<I>` shall define the nested type named `iterator_category` as a synonym for `input_iterator_tag`, otherwise as a synonym for `weak_input_iterator_tag`.

2. Let `R` be `reference_t<I>`. If `is_reference<R>::value` is true, the template specialization `move_iterator<I>` shall define the nested type named `reference` as a synonym for `remove_reference_t<R>&&`, otherwise as a synonym for `R`.

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]
24.8.3.3.2 move_iterator::operator=

template <class WeakInputIterator U>
requires ConvertibleTo<U, I>()
mov_iterator& operator=(const move_iterator<U>& u);

1 Effects: Assigns u.base() to u.current to current.
2 Requires: U shall be convertible to Iterator.

24.8.3.3.3 move_iterator conversion

Iterator base() const;
1 Returns: current.

24.8.3.3.4 move_iterator::operator*

reference operator*() const;
1 Returns: Effects: Equivalent to std::move static_cast<reference>(*current).

24.8.3.3.5 move_iterator::operator->

pointer operator->() const;
1 Returns: current.

24.8.3.3.6 move_iterator::operator++

move_iterator& operator++();
1 Effects: Equivalent to ++current.
2 Returns: *this.

move_iterator operator++(int);
3 Effects: Equivalent to

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

24.8.3.3.7 move_iterator::operator--

move_iterator& operator--();
requires BidirectionalIterator<I>();
1 Effects: Equivalent to --current.
2 Returns: *this.

move_iterator operator--(int);
requires BidirectionalIterator<I>();
3 Effects: Equivalent to

    move_iterator tmp = *this;
    --current;
    return tmp;
move_iterator operator+(difference_type n) const
<requires RandomAccessIterator<I>();

Returns: Effects: Equivalent to move_iterator(current + n).

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

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

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

Returns: Effects: Equivalent to move_iterator(current - n).

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

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

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

Returns: Effects: Equivalent to std::move static_cast<reference>(current[n]).

template <class Iterator1 InputIterator I1, class Iterator2 InputIterator I2>
<requires EqualityComparable<I1, I2>();
bool operator==(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);

Returns: Effects: Equivalent to x.base().current == y.base().current.

template <class Iterator1 InputIterator I1, class Iterator2 InputIterator I2>
<requires EqualityComparable<I1, I2>();
bool operator!=(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);

Returns: Effects: Equivalent to !(x == y).

template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
<requires StrictTotallyOrdered<I1, I2>();
bool operator<(const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);

Returns: Effects: Equivalent to x.base().current < y.base().current.
template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator<=(
    const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);

4 Returns: Effects: Equivalent to !(y < x).

template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator>(
    const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);

5 Returns: Effects: Equivalent to y < x.

template <class Iterator1 RandomAccessIterator I1, class Iterator2 RandomAccessIterator I2>
requires StrictTotallyOrdered<I1, I2>()
bool operator>=(
    const move_iterator<Iterator1 I1>& x, const move_iterator<Iterator2 I2>& y);

6 Returns: Effects: Equivalent to !(x < y).

24.8.3.3.14 move_iterator non-member functions

template <class Iterator1 WeakInputIterator I1, class Iterator2 WeakInputIterator I2>
requires SizedIteratorRange<I2, I1>()
auto DifferenceType<I2>difference_type_t<I2> operator-(
    const move_iterator<Iterator1 I1>& x,
    const move_iterator<Iterator2 I2>& y) noexcept(y.base() - x.base());

1 Returns: Effects: Equivalent to x.base().current - y.base().current.

template <class Iterator RandomAccessIterator I>
move_iterator<Iterator I>
operator+(typename move_iterator<Iterator>::difference_type
    difference_type_t<I> n,
    const move_iterator<Iterator I>& x);

2 Returns: Effects: Equivalent to x + n.

template <class Iterator WeakInputIterator I>
move_iterator<Iterator I> make_move_iterator(
    Iterator I i);

3 Returns: move_iterator<Iterator I>(i).

24.8.4 Common iterators

1 Class template common_iterator is an iterator/sentinel adaptor that is capable of representing a non-
bounded range of elements (where the types of the iterator and sentinel differ) as a bounded range (where
they are the same). It does this by holding either an iterator or a sentinel, and implementing the equality
comparison operators appropriately.

2 [Note: The common_iterator type is useful for interfacing with legacy code that expects the begin and
end of a range to have the same type, and for use in common_type specializations that are required to make
iterator/sentinel pairs satisfy the EqualityComparable concept. — end note]

3 [Example:
template<class ForwardIterator>
void fun(ForwardIterator begin, ForwardIterator end);

list<int> s;
// populate the list s
using CI =
  common_iterator<counted_iterator<list<int>::iterator>,
  default_sentinel>;
// call fun on a range of 10 ints
fun(CI(make_counted_iterator(s.begin(), 10)),
  CI(default_sentinel()));

— end example ]

24.8.4.1 Class template common_iterator

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

namespace

concept bool __WeaklyEqualityComparable =
  EqualityComparable<A>() && EqualityComparable<B>() &&
  requires(const A a, const B b) {
    {a==b} -> Boolean;
    {a!=b} -> Boolean;
    {b==a} -> Boolean;
    {b!=a} -> Boolean;
  };

concept bool __WeakSentinel =
  Iterator<I>() && Regular<S>() &&
  __WeaklyEqualityComparable<I, S>;

constexpr bool _fwd_iter = false; // exposition only

class common_iterator {

public:
  using difference_type = DifferenceType_t<I>;
  using value_type = ValueType_t<I>;
  using iterator_category = conditional_t<_fwd_iter,
    std::forward_iterator_tag,
    std::input_iterator_tag>;
  using reference = ReferenceType_t<I>;

  common_iterator();
  common_iterator(I i);
  common_iterator(S s);

  template <InputIterator class U, __WeakSentinel class V>
  requiresConvertibleTo<U, I>() && ConvertibleTo<V, S>()
  common_iterator(const common_iterator<U, V>& u);

}}  // namespace ranges
} // namespace experimental
} // namespace std

§ 24.8.4.1 80
common_iterator& operator=(const common_iterator<U, V>& u);

~common_iterator();

see below operator*();

reference see below operator*() const;

see below operator->() const requires Readable<I>();

common_iterator& operator++();

common_iterator operator++(int);

private:

bool is_sentinel; // exposition only
I iter; // exposition only
S sent; // exposition only

};

template <InputIterator I, class S>
struct iterator_category<common_iterator<I, S>> {
    using type = see below;
};

// exposition only

template <WeaklyIncrementable I1, WeaklyIncrementable I2>
using __common_difference_type_t = // exposition only
    common_type_t<difference_type_t<I1>, difference_type_t<I2>>;

// exposition only

template <class I1, class S1, class I2, class S2>
concept bool __CommonSizedIteratorRange() { // exposition only
    return SizedIteratorRange<I1, I1>() && SizedIteratorRange<I2, I2>() &&
        requires (const I1 i1, const S1 s1, const I2 i2, const S2 s2) {
        {i1-i2}->__common_difference_type_t<I1, I2>; {i2-i1}->__common_difference_type_t<I1, I2>;
        {i1-s2}->__common_difference_type_t<I1, I2>; {s2-i1}->__common_difference_type_t<I1, I2>;
        {i2-s1}->__common_difference_type_t<I1, I2>; {s1-i2}->__common_difference_type_t<I1, I2>;
    };
};

// exposition only

template <InputIterator class I1, __WeakSentinel<i1> class S1,
    InputIterator class I2, __WeakSentinel<i2> class S2>
    requires EqualityComparable<I1, I2>() && __WeaklyEqualityComparable<I1, S2> &&
    __WeaklyEqualityComparable<I2, S1>
bool operator==(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

// exposition only

template <InputIterator class I1, __WeakSentinel<i1> class S1,
    InputIterator class I2, __WeakSentinel<i2> class S2>
    requires EqualityComparable<I1, I2>() && __WeaklyEqualityComparable<I1, S2> &&
    __WeaklyEqualityComparable<I2, S1>
bool operator!=(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

// exposition only

template <InputIterator class I1, __WeakSentinel<i1> class S1,
    InputIterator class I2, __WeakSentinel<i2> class S2>
    requires SizedIteratorRange<I1, I1>() && SizedIteratorRange<I2, I2>() &&
    requires (const I1 i1, const S1 s1, const I2 i2, const S2 s2) {
        {i1-i2)->DifferenceType<T2>; {i2-i1)->DifferenceType<T2>;
        {i1-s2)->DifferenceType<T2>; {s2-i1)->DifferenceType<T2>;
        

§ 24.8.4.1
\{i2-s1\} \rightarrow \text{DifferenceType}\{I2\}; \{s1-i2\} \rightarrow \text{DifferenceType}\{I2\};

\text{requires} \text{CommonSizedIteratorRange}\{I1, S1, I2, S2\}()
\text{DifferenceType}\{I2\} \text{common_difference_type_t}\{I1, I2\} \text{operator-}(
\text{const common_iterator}\{I1, S1\}& x, \text{const common_iterator}\{I2, S2\}& y);
}}

1 For a type \(I\) that satisfies InputIterator and a type \(S\), the partial specialization \text{iterator_category}<\text{common_iterator}\{I, S\}>> shall define a nested type named \text{name} as follows:

(1.1) — If \(I\) satisfies ForwardIterator, then \text{iterator_category}<\text{common_iterator}\{I, S\}>::\text{type} shall be a synonym for \text{forward_iterator_tag}.

(1.2) — Otherwise, \text{iterator_category}<\text{common_iterator}\{I, S\}>::\text{type} shall be a synonym for \text{input_iterator_tag}.

2 [Note: The use of the expository \text{__WeaklyEqualityComparable} and \text{__WeakSentinel} concepts is to avoid the self-referential requirements that would happen if parameters \(I\) and \(S\) use \text{common_iterator}\{I, S\} as their common type. — end note]

3 [Note: The ad hoc constraints on \text{common_iterator}'s operator- exist for the same reason. — end note]

4 [Note: It is unspecified whether \text{common_iterator}'s members \text{iter} and \text{sent} have distinct addresses or not. — end note]

24.8.4.2 \text{common_iterator} operations

24.8.4.2.1 \text{common_iterator} constructors

\text{common_iterator}();
1 \text{Effects:} Constructs a \text{common_iterator}, value-initializing \text{is_sentinel} and \text{iter}. 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 \(I\).

\text{Remarks:} It is unspecified whether any initialization is performed for \text{sent}.

\text{common_iterator}(I i);
3 \text{Effects:} Constructs a \text{common_iterator}, initializing \text{is_sentinel} with \text{false} and \text{iter} with \(i\).

\text{Remarks:} It is unspecified whether any initialization is performed for \text{sent}.

\text{common_iterator}(S s);
5 \text{Effects:} Constructs a \text{common_iterator}, initializing \text{is_sentinel} with \text{true} and \text{sent} with \(s\).

\text{Remarks:} It is unspecified whether any initialization is performed for \text{iter}.

\text{template}<\text{InputIterator class } U, \text{__WeakSentinel class } V>
\text{requires } \text{ConvertibleTo}\{U, I\}() \&\& \text{ConvertibleTo}\{V, S\}()
\text{common_iterator}(\text{const common_iterator}\{U, V\}& u);
7 \text{Effects:} Constructs a \text{common_iterator}, initializing \text{is_sentinel} with \(u.\text{is_sentinel}\).

(7.1) — If \(u.\text{is_sentinel}\) is \text{true}, \text{sent} is initialized with \(u.\text{sent}\).

(7.2) — If \(u.\text{is_sentinel}\) is \text{false}, \text{iter} is initialized with \(u.\text{iter}\).

\text{Remarks:}

(7.3) — If \(u.\text{is_sentinel}\) is \text{true}, it is unspecified whether any initialization is performed for \text{iter}.

(7.4) — If \(u.\text{is_sentinel}\) is \text{false}, it is unspecified whether any initialization is performed for \text{sent}.

§ 24.8.4.2.1
24.8.4.2.2  common_iterator::operator=

[common.iter.op=]

template <InputIterator class U, WeakSentinel class V>
requiresConvertibleTo< U, I>() && ConvertibleTo< V, S>()
common_iterator& operator=(const common_iterator< U, V>& u);

Effects: Assigns u.is_sentinel to is_sentinel.

(1.1) If u.is_sentinel is true, assigns u.sent to sent.
(1.2) If u.is_sentinel is false, assigns u.iter to iter.

Remarks:

(1.3) If u.is_sentinel is true, it is unspecified whether any operation is performed on iter.
(1.4) If u.is_sentinel is false, it is unspecified whether any operation is performed on sent.

Returns: *this
~common_iterator();

Effects: Runs the destructor(s) for Destroys any members that are currently initialized.

24.8.4.2.3  common_iterator::operator*

[common.iter.op.star]

decltype(auto) operator*();
reference decltype(auto) operator*() const;

Requires: !is_sentinel

Returns: Effects: Equivalent to *iter.

see below operator->() const requires Readable<I>();

Requires: !is_sentinel

Effects: Given an object i of type I

(4.1) if I is a pointer type or if the expression i.operator->() is a well-formed expression, this function returns iter.
(4.2) Otherwise, if the expression *iter is an glvalue, this function is equivalent to addressof(*iter).
(4.3) Otherwise, this function returns a proxy object of an unspecified type equivalent to the following:

class proxy {
    // exposition only
    value_type_t<I> keep_; // exposition only
    proxy(reference_t<I>&& x)
        : keep_(std::move(x)) {}
    public:
        const value_type_t<I>* operator->() const {
            return addressof(keep_);
        }
};

that is initialized with *iter.
24.8.4.2.4 common_iterator::operator++

common_iterator& operator++();

1  Requires: !is_sentinel
2  Effects: ++iter.
3  Returns: *this.

common_iterator operator++(int);

4  Requires: !is_sentinel
5  Effects: Equivalent to
   common_iterator tmp = *this;
   ++iter;
   return tmp;

24.8.4.2.5 common_iterator comparisons

template <InputIterator class I1, __WeakSentinel class S1,
          InputIterator class I2, __WeakSentinel class S2>
requires EqualityComparable<I1, I2>() && __WeaklyEqualityComparable<I1, S2> &&
__WeaklyEqualityComparable<I2, S1>
bool operator==(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

1  Returns: Effects: Equivalent to
   x.is_sentinel ?
   (y.is_sentinel || y.iter == x.sent) :
   (y.is_sentinel ?
    x.iter == y.sent :
    x.iter == y.iter);

template <InputIterator class I1, __WeakSentinel class S1,
          InputIterator class I2, __WeakSentinel class S2>
requires EqualityComparable<I1, I2>() && __WeaklyEqualityComparable<I1, S2> &&
__WeaklyEqualityComparable<I2, S1>
bool operator!=(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

2  Returns: Effects: Equivalent to !x == y.

template <InputIterator class I1, __WeakSentinel class S1,
          InputIterator class I2, __WeakSentinel class S2>
requires SizedIteratorRange<I1, I1>() && SizedIteratorRange<I2, I2>() &&
requires (const I1 i1, const S1 s1, const I2 i2, const S2 s2) &
   (i1-i2)->DifferenceType<I2>(), (i2-i1)->DifferenceType<I2>,
   (i1-s1)->DifferenceType<I2>(), (s2-i1)->DifferenceType<I2>,
   (i2-s1)->DifferenceType<I2>(), (s1-i2)->DifferenceType<I2>),

3  Returns: Effects: Equivalent to
   DifferenceTypeT2a __common_difference_type_t<I1, I2> operator-(
   const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
x.is_sentinel ?
  (y.is_sentinel ? 0 : x.sent - y.iter) :
(y.is_sentinel ?
  x.iter - y.sent :
  x.iter - y.iter);  

24.8.5 Default sentinels

24.8.5.1 Class default_sentinel

namespace std { namespace experimental { namespace ranges {\
  namespace _v1 {\
    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 (24.8.6.1)).

24.8.5.2 default_sentinel operations

24.8.5.2.1 default_sentinel comparisons

constexpr bool operator==(default_sentinel x, default_sentinel y) noexcept;

Returns: true

constexpr bool operator!=(default_sentinel x, default_sentinel y) noexcept;

Returns: false

constexpr bool operator<(default_sentinel x, default_sentinel y) noexcept;

Returns: false

constexpr bool operator<=(default_sentinel x, default_sentinel y) noexcept;

Returns: true

constexpr bool operator>(default_sentinel x, default_sentinel y) noexcept;

Returns: false

constexpr bool operator>=(default_sentinel x, default_sentinel y) noexcept;

Returns: true

24.8.5.2.2 default_sentinel non-member functions

constexpr ptrdiff_t operator-(default_sentinel x, default_sentinel y) noexcept;

Returns: 0

24.8.6 Counted iterators

Class template counted_iterator is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of its distance from its starting position. It can be used together with class default_sentinel in calls to generic algorithms to operate on a range of $N$ elements starting at a given position without needing to know the end position a priori.

Example:

list<string> s;
// populate the list s
vector<string> 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 `next(i1.base(), i1.count())` and `next(i2.base(), i2.count())` refer to the same (possibly past-the-end) element.

### 24.8.6.1 Class template `counted_iterator`

```cpp
namespace std { namespace experimental { namespace ranges_v1 {
    template <WeakIterator I>
    class counted_iterator {
    public:
        using iterator_type = I;
        using difference_type = DifferenceType
                               difference_type_t<I>;
        using reference = ReferenceType<I>;

        counted_iterator();
        counted_iterator(I x, DifferenceType
                        difference_type_t<I> n);
        template <WeakIterator U>
        requires ConvertibleTo<U, I>()
        counted_iterator(const counted_iterator<U>& u);
        template <WeakIterator U>
        requires ConvertibleTo<U, I>()
        counted_iterator& operator=(const counted_iterator<U>& u);
        I base() const;
        DifferenceType
difference_type_t<I> count() const;

        counted_iterator& operator++();
        counted_iterator operator++(int);
        counted_iterator& operator--()
        requires BidirectionalIterator<I>();
        counted_iterator operator--(int)
        requires BidirectionalIterator<I>();
        counted_iterator operator+ (difference_type n) const
        requires RandomAccessIterator<I>();
        counted_iterator& operator+=(difference_type n)
        requires RandomAccessIterator<I>();
        counted_iterator operator- (difference_type n) const
        requires RandomAccessIterator<I>();
        counted_iterator& operator-=(difference_type n)
        requires RandomAccessIterator<I>();

    private:
        I current; // exposition only
        DifferenceType
difference_type_t<I> cnt; // exposition only
    };

    template <WeakInputIterator I>
    struct value_type<counted_iterator<I>> : value_type<I> { };}
```

§ 24.8.6.1
struct iterator_category<counted_iterator<I>> {
    using type = input_iterator_tag;
};

template <ForwardIterator I>
struct iterator_category<counted_iterator<I>> : iterator_category<I> { }

// Other code follows...
template <WeakIterator I>
bool operator>(
    default_sentinel x, const counted_iterator<I>& y);

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

template <WeakIterator I>
bool operator>(
    const counted_iterator<I>& x, default_sentinel y);

template <WeakIterator I>
bool operator>(
    default_sentinel x, const counted_iterator<I>& y);

template <WeakIterator I1, WeakIterator I2>
requires Common<I1, I2>()
DifferenceType<I2>
common_type_t<difference_type_t<I1>, difference_type_t<I2>>
operator-(const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

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

template <WeakIterator I>
counted_iterator<I>
make_counted_iterator(I i,
    DifferenceType<
    difference_type_t<I>,
    difference_type_t<I>> n);

template <WeakIterator I>
void advance(counted_iterator<I>& i,
    DifferenceType<
    difference_type_t<I>,
    difference_type_t<I>> n);

}}}

24.8.6.2 counted_iterator operations
24.8.6.2.1 counted_iterator constructors

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.

counted_iterator(I i, DifferenceType<
    difference_type_t<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 <WeakIterator U>
requiresConvertibleTo<U, I>()
counted_iterator(const counted_iterator<U>& u);

4 Effects: Constructs a counted_iterator, initializing current with u.base().current and cnt with u.count().cnt.
24.8.6.2.2 counted_iterator::operator=

```cpp
template <WeakIterator U>
    requiresConvertibleTo(U, I())
counted_iterator & operator=(const counted_iterator<U> & u);
```

**Effects:** Assigns \( u.\text{base}() \) current to current and \( u.\text{count}() \) cnt to cnt.

24.8.6.2.3 counted_iterator conversion

```cpp
I base() const;
```

**Returns:** current.

24.8.6.2.4 counted_iterator count

```cpp
DifferenceType difference_type_t<I> count() const;
```

**Returns:** cnt.

24.8.6.2.5 counted_iterator::operator*

```cpp
decaytype(auto) operator*();
    reference decaytype(auto) operator*() const;
```

**Returns:** Equivalent to \(*\)current.

24.8.6.2.6 counted_iterator::operator++

```cpp
counted_iterator & operator++();
```

**Requires:** cnt > 0

**Effects:** Equivalent to

\[
++\text{current};
--\text{cnt};
\]

**Returns:** *this.

```cpp
counted_iterator operator++(int);
```

**Requires:** cnt > 0

**Effects:** Equivalent to

\[
\text{counted_iterator tmp = *this;}\]
\[
++\text{current};
--\text{cnt};
\]

**Returns:** *this.

24.8.6.2.7 counted_iterator::operator--

```cpp
counted_iterator & operator--();
```

**Requires:** BidirectionalIterator\(<\)I>()

**Effects:** Equivalent to

\[
--\text{current};
++\text{cnt};
\]

**Returns:** *this.
counted_iterator operator--(int)
   requires BidirectionalIterator<It>();

   Effects: Equivalent to
   counted_iterator tmp = *this;
   --current;
   ++cnt;
   return tmp;

24.8.6.2.8 counted_iterator::operator+
               [counted.iter.op.+]

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

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

24.8.6.2.9 counted_iterator::operator++
              [counted.iter.op.++]

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

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

24.8.6.2.10 counted_iterator::operator-
               [counted.iter.op.-]

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

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

24.8.6.2.11 counted_iterator::operator-=
              [counted.iter.op.-=]

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

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

24.8.6.2.12 counted_iterator::operator[]
               [counted.iter.op.index]

   unspecified decltype(auto) operator[](difference_type n) const
   requires RandomAccessIterator<It>();

   Requires: n <= cnt
   Returns: Effects: Equivalent to current[n].
24.8.6.2.13 counted_iterator comparisons

[ counted.iter.op.comp ]

template <WeakIterator I1, WeakIterator I2>
  requires Common<I1, I2>()
  bool operator==(const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  Requires: x and y shall refer to elements of the same sequence (24.8.6).

  Returns: Effects: Equivalent to \( \text{x.count()}_\text{cnt} \) \( \text{==} \) \( \text{y.count()}_\text{cnt} \).

template <WeakIterator I>
  bool operator==(const counted_iterator<I>& x, default_sentinel y);
  Returns: Effects: Equivalent to \( \text{x.count()}_\text{cnt} \) \( \text{==} \) \( \text{0} \).

template <WeakIterator I>
  bool operator==(default_sentinel x, const counted_iterator<I>& y);
  Returns: Effects: Equivalent to \( \text{y.count()}_\text{cnt} \) \( \text{==} \) \( \text{0} \).

template <WeakIterator I1, WeakIterator I2>
  requires Common<I1, I2>()
  bool operator!=(const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <WeakIterator I>
  bool operator!=(const counted_iterator<I>& x, const counted_iterator<I>& y);
  template <WeakIterator I>
  bool operator!=(const counted_iterator<I>& x, default_sentinel y);
  template <WeakIterator I>
  bool operator!=(default_sentinel x, const counted_iterator<I>& y);
  Requires: For the first overload, x and y shall refer to elements of the same sequence (24.8.6).
  Returns: Effects: Equivalent to \( \text{!(x == y)} \).

  template <WeakIterator I1, WeakIterator I2>
  requires Common<I1, I2>()
  bool operator<(const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  Requires: x and y shall refer to elements of the same sequence (24.8.6).
  Returns: Effects: Equivalent to \( \text{y.count()}_\text{cnt} \) \( \text{<} \) \( \text{x.count()}_\text{cnt} \).
  Remark: The argument order in the Effects clause are reversed because \( \text{cnt} \) counts down, not up.

  template <WeakIterator I>
  bool operator<(const counted_iterator<I>& x, default_sentinel y);
  Returns: Effects: Equivalent to \( \text{x.count()}_\text{cnt} \) \( \text{!=} \) \( \text{0} \).

  template <WeakIterator I>
  bool operator<(default_sentinel x, const counted_iterator<I>& y);
  Returns: \( \text{false} \).

§ 24.8.6.2.13
template <WeakIterator I1, WeakIterator I2>
    requires Common<I1, I2>()
    bool operator<=(
        const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

Requires: For the first overload, \(x\) and \(y\) shall refer to elements of the same sequence (24.8.6).

Returns: Equivalent to \(!(y < x)\).

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

template <WeakIterator I>
    bool operator>(
        const counted_iterator<I>& x, default_sentinel y);

template <WeakIterator I>
    bool operator>(
        default_sentinel x, const counted_iterator<I>& y);

Requires: For the first overload, \(x\) and \(y\) shall refer to elements of the same sequence (24.8.6).

Returns: Equivalent to \(y < x\).

template <WeakIterator I1, WeakIterator I2>
    requires Common<I1, I2>()
    bool operator==( 
        const counted_iterator<I1>& x, const counted_iterator<I2>& y);

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

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

Requires: For the first overload, \(x\) and \(y\) shall refer to elements of the same sequence (24.8.6).

Returns: Equivalent to \(!(x < y)\).

24.8.6.2.14 counted_iterator non-member functions [counted.iter.nonmember]

template <WeakIterator I1, WeakIterator I2>
    requires Common<I1, I2>()
    DifferenceType<I2>
        common_type_t<
            difference_type_t<I1>,
            difference_type_t<I2>>
        operator- (const counted_iterator<I1>& x, const counted_iterator<I2>& y);

Requires: For the first overload, \(x\) and \(y\) shall refer to elements of the same sequence (24.8.6).

Returns: Equivalent to \(y.count() - x.count()\).

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

Returns: Effects: Equivalent to \(-x\).\(\text{cnt}\).

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

    Requires: \(n \leq x\).\(\text{cnt}\).

    Returns: Effects: Equivalent to \(x + n\).
```

```
template <WeakIterator I>
counted_iterator<I> make_counted_iterator(I i,
    DifferenceType difference_type_t<I> n);

    Requires: \(n \geq 0\).

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

```
template <WeakIterator I>
    void advance(counted_iterator<I>& i,
        DifferenceType difference_type_t<I> n);

    Requires: \(n \leq i\).\(\text{cnt}\).

    Effects:
        \(i = \text{make_counted_iterator(next(i.\text{base()}.\text{current}, n), i.\text{count()}.\text{cnt} - n)}\);
```

### 24.8.6.3 Specializations of \texttt{common\_type}

\texttt{namespace std {}
\begin{verbatim}
    template<experimental::ranges_v1::WeakIterator I>
        struct common_type<experimental::ranges_v1::counted_iterator<I>,
            experimental::ranges_v1::default_sentinel> {  
            using type = experimental::ranges_v1::common_iterator<
                experimental::ranges_v1::counted_iterator<I>,
                experimental::ranges_v1::default_sentinel;  
            };  

    template<experimental::ranges_v1::WeakIterator I>
        struct common_type<experimental::ranges_v1::default_sentinel,
            experimental::ranges_v1::counted_iterator<I>> {  
            using type = experimental::ranges_v1::common_iterator<
                experimental::ranges_v1::counted_iterator<I>,
                experimental::ranges_v1::default_sentinel;  
            };  
\end{verbatim}
\texttt{}}

\texttt{}}

\texttt{\[ Note: By specializing \texttt{common\_type} this way, \texttt{counted\_iterator} and \texttt{default\_sentinel} can satisfy the Common requirement of the \texttt{EqualityComparable} concept. \textendnote \]}

### 24.8.7 Dangling wrapper

\texttt{\[dangling.wrappers\]}

#### 24.8.7.1 Class template dangling

\texttt{\[dangling.wrap\]}

Class template \texttt{dangling} is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {

template <CopyConstructible T>
class dangling {
public:
    dangling() requires DefaultConstructible<T>();
    dangling(T t);
    T get_unsafe() const;
private:
    T value; // exposition only
};

template <Range R>
using safe_iterator_t =
    conditional_t<is_lvalue_reference<R>::value,
        IteratorType_iterator_t<R>,
        dangling<IteratorType_iterator_t<R>>;}}

24.8.7.2 dangling operations [dangling.wrap.ops]
24.8.7.2.1 dangling constructors [dangling.wrap.op.const]

dangling() requires DefaultConstructible<T>();
1
    Effects: Constructs a dangling, value-initializing value.

dangling(T t);
2
    Effects: Constructs a dangling, initializing value with t.

24.8.7.2.2 dangling::get_unsafe [dangling.wrap.op.get]

T get_unsafe() const;
1
    Returns: value.

24.8.8 Unreachable sentinel [unreachable.sentinels]
24.8.8.1 Class unreachable [unreachable.sentinel]

1 Class unreachable is a sentinel type that can be used with any Iterator to denote an infinite range. Comparing an iterator for equality with an object of type unreachable always returns false.

[Example:

char* p;
    // set p to point to a character buffer containing newlines
char* nl = find(p, unreachable(), '\n');

Provided a newline character really exists in the buffer, the use of unreachable above potentially makes the call to find more efficient since the loop test against the sentinel does not require a conditional branch.
—end example]

namespace std { namespace experimental { namespace ranges_v1 { 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;
    constexpr bool operator==(unreachable, unreachable) noexcept;

§ 24.8.8.1
template <Iterator I>
constexpr bool operator!=(const I&, unreachable) noexcept;
template <Iterator I>
constexpr bool operator!=(unreachable, const I&) noexcept;
constexpr bool operator!=(unreachable, unreachable) noexcept;

template <Iterator I>
constexpr bool operator<(const I&, unreachable) noexcept;
template <Iterator I>
constexpr bool operator<(unreachable, const I&) noexcept;
constexpr bool operator<(unreachable, unreachable) noexcept;

template <Iterator I>
constexpr bool operator<=(const I&, unreachable) noexcept;
template <Iterator I>
constexpr bool operator<=(unreachable, const I&) noexcept;
constexpr bool operator<=(unreachable, unreachable) noexcept;

template <Iterator I>
constexpr bool operator>(const I&, unreachable) noexcept;
template <Iterator I>
constexpr bool operator>(unreachable, const I&) noexcept;
constexpr bool operator>(unreachable, unreachable) noexcept;

template <Iterator I>
constexpr bool operator>=(const I&, unreachable) noexcept;
template <Iterator I>
constexpr bool operator>=(unreachable, const I&) noexcept;
constexpr bool operator>=(unreachable, unreachable) noexcept;

}}}}

24.8.8.2 unreachable operations

24.8.8.2.1 operator==

template <Iterator I>
constexpr bool operator==(const I&, unreachable) noexcept;
template <Iterator I>
constexpr bool operator==(unreachable, const I&) noexcept;
constexpr bool operator==(unreachable, unreachable) noexcept;

Returns: false.

Returns: true.

24.8.8.2.2 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;
constexpr bool operator!=(unreachable x, unreachable y) noexcept;

Returns: !(x == y)

24.8.8.2.3 operator<

template <Iterator I>
constexpr bool operator<(const I&, unreachable) noexcept;
Returns: true.

```cpp
template <Iterator I>
constexpr bool operator<(unreachable, const I&) noexcept;
constexpr bool operator<(unreachable, unreachable) noexcept;
```

Returns: false.

24.8.8.2.4 operator<=

```cpp
template <Iterator I>
constexpr bool operator<=(const I& x, unreachable y) noexcept;
```

24.8.8.2.5 operator>

```cpp
template <Iterator I>
constexpr bool operator>(const I& x, unreachable y) noexcept;
```

24.8.8.2.6 operator>=

```cpp
template <Iterator I>
constexpr bool operator>=(const I& x, unreachable y) noexcept;
```

1 Returns: !(y < x).

24.9 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]

24.9.1 Class template istream_iterator

[istream.iterator]

The class template istream_iterator is an input iterator (24.2.14) 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.

namespace std {
    namespace experimental {
        namespace ranges_v1 {
            template <class T, class charT = char, class traits = char_traits<charT>,
                class Distance = ptrdiff_t>
            class istream_iterator<
                T, charT, traits, Distance> {
                public:
                    typedef input_iterator_tag iterator_category;
                    typedef Distance difference_type;
                    typedef T value_type;
                    typedef const T& reference;
                    typedef const T* pointer;
                    template basic_istream<charT, traits> istream_type;
                    see below istream_iterator();
                    see below 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<T, charT, traits, Distance>& operator++();
                    istream_iterator<T, charT, traits, Distance> operator++(int);

                private:
                    basic_istream<charT, traits>* in_stream; // exposition only
                    T value; // exposition only
            };

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

        };
    }
};

§ 24.9.1


```cpp
bool operator==(default_sentinel x, const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T, charT, traits, Distance>& x, default_sentinel y);
template <class T, class charT, class traits, class Distance>
bool operator!=(const istream_iterator<T, charT, traits, Distance>& x, const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
bool operator!=(default_sentinel x, const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
bool operator!=(const istream_iterator<T, charT, traits, Distance>& x, default_sentinel y);
}
```
istream_iterator<T, charT, traits, Distance> operator++(int);

6 Requires: in_stream != nullptr.

7 Effects:

istream_iterator<T, charT, traits, Distance> tmp = *this;
*in_stream >> value;
return tmp;

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

8 Returns: x.in_stream == y.in_stream.

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

9 Returns: nullptr == y.in_stream.

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

10 Returns: x.in_stream == nullptr.

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

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

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

11 Returns: !(x == y)

24.9.2 Class template ostream_iterator

1 ostream_iterator writes (using operator<<) successive elements onto the output stream from which it was constructed. If it was constructed with charT* as a constructor argument, this string, called a delimiter string, is written to the stream after every T is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

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

2 ostream_iterator is defined as:

namespace std {
    namespace experimental {
    namespace ranges_v1 {
    inline namespace v1 {
    template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator<
        public_iterator<output_iterator_tag, void, void, void, void>,
    public:
        typedef output_iterator_tag_iterator_category;
        typedef ptrdiff_t difference_type;

§ 24.9.2

99
```cpp
typedef char char_type;
typedef traits traits_type;
typedef basic_ostream<charT, traits> ostream_type;
constexpr ostream_iterator() noexcept;
ostream_iterator(ostream_type& s) noexcept;
ostream_iterator(const charT* delimiter) noexcept;
~ostream_iterator();
ostream_iterator<T, charT, traits>& operator=(const T& value);
ostream_iterator<T, charT, traits>& operator*();
ostream_iterator<T, charT, traits>& operator++();
ostream_iterator<T, charT, traits>& operator++(int);

private:
  basic_ostream<charT, traits>* out_stream;  // exposition only
  const charT* delim;  // exposition only
};
24.9.3 Class template istreambuf_iterator

The class template `istreambuf_iterator` defines an input iterator (24.2.14) that reads successive characters from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. [Note: `operator->` may return a proxy. — end note] 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_ value is returned. It is impossible to assign a character via an input iterator.

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
    template<class charT, class traits = char_traits<charT>>
    class istreambuf_iterator
    {
        public:
            typedef input_iterator_tag iterator_category;
            typedef charT value_type;
            typedef typename traits::off_type difference_type;
            typedef charT reference;
            typedef unspecified pointer;
            typedef charT char_type;
            typedef traits traits_type;
            typedef basic_streambuf<charT, traits> streambuf_type;
            typedef basic_istream<charT, traits> istream_type;

            class proxy;      // exposition only

            constexpr istreambuf_iterator() noexcept;
            constexpr istreambuf_iterator(default_sentinel) noexcept;
            istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
            ~istreambuf_iterator() = default;
            istreambuf_iterator(istream_type& s) noexcept;
            istreambuf_iterator(streambuf_type* s) noexcept;
            istreambuf_iterator(const proxy& p) noexcept;
            charT operator*() const;
            pointer operator->() const;
            istreambuf_iterator<charT, traits>& operator++();
            proxy operator++(int);
            bool equal(const istreambuf_iterator& b) const;

            private:
                streambuf_type* sbuf_;      // exposition only
                }

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

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

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

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

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

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

    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);

```
\begin{verbatim}
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);

24.9.3.1 Class template istreambuf_iterator::proxy

namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {
    template <class charT, class traits = char_traits<charT> >
    class istreambuf_iterator<charT, traits>::proxy { // exposition only
        charT keep_;
        basic_streambuf<charT, traits>* sbuf_;
        proxy(charT c, basic_streambuf<charT, traits>* sbuf)
          : keep_(c), sbuf_(sbuf) { }
    public:
        charT operator*() { return keep_; }
    };
}}}

1 Class istreambuf_iterator<charT, traits>::proxy is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class istreambuf_iterator<charT, traits>::proxy provides a temporary placeholder as the return value of the post-increment operator (operator++). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

24.9.3.2 istreambuf_iterator constructors

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

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

istreambuf_iterator(basic_istream<charT, traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT, traits>* s) noexcept;

2 Effects: Constructs an istreambuf_iterator<> that uses the basic_streambuf<> object *(s.rdbuf()), or *s, respectively. Constructs an end-of-stream iterator if s.rdbuf() is null.

istreambuf_iterator(const proxy& p) noexcept;

3 Effects: Constructs a istreambuf_iterator<> that uses the basic_streambuf<> object pointed to by the proxy object’s constructor argument p.

24.9.3.3 istreambuf_iterator::operator*

charT operator*() const

1 Returns: The character obtained via the streambuf member sbuf_->sgetc().
\end{verbatim}
24.9.3.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```cpp
istreambuf_iterator<
    charT,
    traits>&
operator++();
```

1. **Effects:** Equivalent to `sbuf_->sbumpc()`.

2. **Returns:** `*this`.

proxy `istreambuf_iterator<
    charT,
    traits>::operator++(int);`

3. **Returns:** Equivalent to `proxy(sbuf_->sbumpc(), sbuf_)`.

24.9.3.5 `istreambuf_iterator::equal` [istreambuf.iterator::equal]

```cpp
bool equal(const istreambuf_iterator<
    charT,
    traits>& b) const;
```

1. **Returns:** `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

24.9.3.6 `operator==` [istreambuf.iterator::op==]

```cpp
template <class charT, class traits>
bool operator==(const istreambuf_iterator<
    charT,
    traits>& a,
    const istreambuf_iterator<
    charT,
    traits>& b);
```

1. **Returns:** Equivalent to `a.equal(b)`.

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

2. **Effects:** Equivalent to `streambuf_iterator<
    charT,
    traits>::{}==(b)).equal(b);`

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

3. **Effects:** Equivalent to `a.equal(istreambuf_iterator<
charT,
traits>::{})`. 

24.9.3.7 `operator!=` [istreambuf.iterator::op!=]

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

1. **Returns:** !a.equal(b). **Effects:** Equivalent to `!(a == b)`.

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

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

2. **Effects:** Equivalent to `a.equal(istreambuf_iterator<
    charT,
    traits>::{})`. 

24.9.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```cpp
namespace std {
    namespace experimental {
        namespace ranges_v1 {
            template <class charT, class traits = char_traits<charT> >
            class ostreambuf_iterator +
            public_iterator<output_iterator_tag, void, void, void, void> {
```

§ 24.9.4
```cpp
typedef output_iterator_tag iterator_category;
typedef ptrdiff_t difference_type;
typedef charT char_type;
typedef traits traits_type;
typedef basic_streambuf<charT, traits> streambuf_type;
typedef basic_ostream<charT, traits> ostream_type;

public:
  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
};
```
Returns: `*this`.

```cpp
tool failed() const noexcept;
```

**Requires:** `sbuf_ != nullptr`.

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

24.10 Range concepts

24.10.1 General

1. This subclause describes components for dealing with ranges of elements.

2. The following subclauses describe range and view requirements, and components for range primitives, pre-defined ranges, and stream ranges, as summarized in Table 5.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.10.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>24.11</td>
<td>&lt;experimental/ranges_v1/iterator&gt;</td>
</tr>
<tr>
<td>24.12</td>
<td>Range primitives</td>
</tr>
</tbody>
</table>

24.10.2 Range requirements

24.10.2.1 In general

1. Ranges are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. It their simplest form, a range object is one on which one can call `begin` and `end` to get an iterator (24.2.11) and a sentinel (24.2.12) or an iterator. To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of ranges.

2. This International Standard defines three fundamental categories of ranges based on the syntax and semantics supported by each: `range`, `sized range` and `view`, as shown in Table 6.

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

3. The `Range` concept requires only that `begin` and `end` return an iterator and a sentinel. [Note: An iterator is a valid sentinel. — end note] The `SizedRange` concept refines `Range` but adds the requirement that the number of elements in the range can be determined in constant time with the `size` function. The `View` concept specifies requirements on an `Range` type with constant-time copy and assign operations.

4. In addition to the three fundamental range categories, this standard defines a number of convenience refinements of `Range` that group together requirements that appear often in the concepts, algorithms, and range adaptors. `Bounded ranges` are ranges for which `begin` and `end` return objects of the same type. `Random access ranges` are ranges for which `begin` returns a model of `iterator` that satisfies `RandomAccessIterator` (24.2.19). The range categories `bidirectional ranges`, `forward ranges`, `input ranges` and `output ranges` are defined similarly. [Note: There are no weak input ranges or weak output ranges...]

§ 24.10.2.1
because of the EqualityComparable requirement on iterators and sentinels. — end note] [Editor's note: TODO: Rethink that because a weak input range would not require (strongly) incrementable iterators.]

### 24.10.2.2 Ranges

The Range concept defines the requirements of a type that allows iteration over its elements by providing a begin iterator and an end iterator or 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>
using IteratorType<iterator_t = decltype(ranges::begin(declval<T&>()))>;

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

template <class T>
concept bool Range() {
    return requires(T t) {
        typename IteratorType<T>,
        typename SentinelType sentinelt_t<T>;
        {begin(t)} -> IteratorType<T>,
        {end(t)} -> SentinelType<T>,
        requires Sentinel<SentineltType<T>, IteratorType<T>>;
    };
}
```

[Note: The semantics of end (24.11.2) ensure that for an expression E, begin(E) is valid whenever end(E) is valid, and the types of the two expressions satisfy Iterator and Sentinel. — end note]

Given an lvalue t of type remove_reference_t<T>, Range<T>() is satisfied if and only if

1. Both begin(t) and end(t) are amortized constant time and non-modifying. [Note: begin(t) and end(t) do not require implicit expression variants. — end note]

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

### 24.10.2.3 Sized ranges

The SizedRange concept describes the requirements of a Range type that knows its size in constant time with the size function.

```cpp
// exposition only
template <class T>
concept bool SizedRangeLike =
    Range<T> &&
    requires(T t) {
        {size(t)} -> Integral,
        requiresConvertible<decltype(size(t))>
        DifferenceType<IteratorType<T>,>
    };

template <class T>
```
struct disable_sized_range;

template <class T>
concept bool SizedRange() {
    __SizedRangeLike<T> && is_sized_range<T>::value &&
    Range<T>() &&
    requires (const remove_reference_t<T>& t) {
        { ranges::size(t) } -> Integral;
        { ranges::size(t) } -> ConvertibleTo<
            DifferenceType<
                difference_type<
                    IteratorType<
                        iterator_t<T>
                >>
            >;
    };
}

2

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

(2.1) size(t) returns the number of elements in t,

(2.2) If IteratorType<
    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 IteratorType iterator type does not model ForwardIterator, for example, size(t) might only be well-defined if evaluated before the first call to begin(t). —end note]

3

For any type T, is_disable_sized_range<T> derives from true_type if T models __SizedRangeLike, and false_type if T is a non-reference cv-unqualified type, and from disable_sized_range<
    remove_cv_t<remove_reference_t<T>>
> otherwise.

4

Users are free to specialize is_disable_sized_range. [Note: Users may want to must specialize is_disable_sized_range so that its member value is true to override the default in the case of accidental conformance for Range types that meet only the syntactic requirements of SizedRange when instantiating a library template that has differing behavior for Range and SizedRange. —end note]

5

[Note: A possible implementation for is_disable_sized_range is given below:]

// exposition only
template <class T>
using __uncvref = remove_cv_t<remove_reference_t<T>>;

template <class R>
struct is_disable_sized_range : false_type
    disable_sized_range<__uncvref<R>>
{
};

template <__SizedRangeLike class R>
requires Same<R, __uncvref<R>>()
struct is_disable_sized_range<R> : true_type false_type
{
};

—end note]

24.10.2.4 Views [ranges.view]

1 The View concept specifies the requirements of a Range type that has constant time copy, move and assignment operators; that is, the cost of these operations is not proportional to the number of elements in the View.

2 [Example: Examples of 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.
— A **Range** type that generates its elements on demand.

A container (23) is not a **View** since copying the container copies the elements, which cannot be done in constant time. — end example]

```cpp
template <class T>
struct enable_view {};

struct view_base {};

// exposition only
template <class T>
constexpr bool __view_predicate = see below;

template <class T>
concept bool View() {
    return Range<T>() &&
    Semiregular<T>() &&
    __view_predicate<T>::value;
}
```

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 the `enable_view` trait to derive from `true_type` or `false_type`. By default, `__view_predicate` uses the following heuristics to determine whether a **Range** type `T` is a **View**:

1. If `T` derives from `view_base`, `__view_predicate<T>::value` is true.
2. If a top-level `const` changes `T`’s `IteratorType`’s `ReferenceType` type, `__view_predicate<T>::value` is false. [Note: Deep `const` ness implies element ownership, whereas shallow `const` ness implies reference semantics. — end note]

For a type `T`, the value of `__view_predicate<T>` shall be:

1. If `enable_view<T>` has a member type `type`, `enable_view<T>::type::value`;
2. If otherwise, if `T` is derived from `view_base`, true;
3. Otherwise, if both `T` and `const T` satisfy **Range** and **ReferenceType**, `reference_t <IteratorType iterator_t<T>>` is not the same type as `reference_t <IteratorType iterator_t<const T>>`, false; [Note: Deep `const` ness implies element ownership, whereas shallow `const` ness implies reference semantics. — end note]
4. Otherwise, true.

[Note: Below is a possible implementation of the `is_view` trait `__view_predicate`.

```cpp
// exposition only
template <class T>
concept bool __ContainerLike =
    Range<T>() &&
    !Same<ReferenceType reference_t <IteratorType iterator_t<T>>,
    ReferenceType reference_t <IteratorType iterator_t<const T>>>();

template <class T>
struct is_view : false_type {};

constexpr bool __view_predicate = true;
```
requires requires { typename enable_view<T>::type; }
constexpr bool __view.predicate<T> = enable_view<T>::type::value;

template <__ContainerLike T>
requires !(DerivedFrom<T, view_base>() ||
    requires { typename enable_view<T>::type; })
constexpr bool __view.predicate<T> = false;

template <Range T>
requires Derived<remove_reference_t<T>, view_base>() ||
    __ContainerLike<remove_reference_t<T>>
struct is_view<T> : __true_type {};

—end note]  

24.10.2.5 Bounded ranges [ranges.bounded]  

1 The BoundedRange concept describes requirements of an Range type for which begin and end return objects of the same type. [Note: The standard containers (23) are models of satisfy BoundedRange. —end note]

template <class T>
concept bool BoundedRange() {  
    return Range<T>() && Same<IteratorType iterator_t<T>, SentinelType sentinel_t<T>>();
}

24.10.2.6 Input ranges [ranges.input]  

1 The InputRange concept describes requirements of an Range type for which begin returns a model of type that satisfies InputIterator (24.2.14).

template <class T>
concept bool InputRange() {  
    return Range<T>() && InputIterator<IteratorType iterator_t<T>>();
}

24.10.2.7 Output ranges [ranges.output]  

1 The OutputRange concept describes requirements of an Range type for which begin returns a type that satisfies OutputIterator (24.2.16).

template <class R, class T>
concept bool OutputRange() {  
    return Range<R>() && OutputIterator<IteratorType iterator_t<R>, T>();
}

24.10.2.8 Forward ranges [ranges.forward]  

1 The ForwardRange concept describes requirements of an InputRange type for which begin returns a model of type that satisfies ForwardIterator (24.2.17).

template <class T>
concept bool ForwardRange() {  
    return InputRange<T>() && ForwardIterator<IteratorType iterator_t<T>>();
}
24.10.2.9 Bidirectional ranges

The BidirectionalRange concept specifies requirements of a ForwardRange type for which begin returns a model of type that satisfies BidirectionalIterator (24.2.18).

```cpp
template <class T>
concept bool BidirectionalRange() {
    return ForwardRange<T>() && BidirectionalIterator<IteratorType iterator_t<T>>();
}
```

24.10.2.10 Random access ranges

The RandomAccessRange concept specifies requirements of a BidirectionalRange type for which begin returns a model of type that satisfies RandomAccessIterator (24.2.19).

```cpp
template <class T>
concept bool RandomAccessRange() {
    return BidirectionalRange<T>() && RandomAccessIterator<IteratorType iterator_t<T>>();
}
```

24.11 Range access

In addition to being available via inclusion of the <iterator> header, the function templates in 24.11 are available when any of the following headers are included: <array>, <deque>, <forward_list>, <list>, <regex>, <set>, <string>, <unordered_map>, <unordered_set>, and <vector>.

```cpp
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
```

Returns: c.begin().

```cpp
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
```

Returns: c.end().

```cpp
template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
```

Returns: array.

```cpp
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
```

Returns: array + N.

```cpp
template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));
```

Returns: std::begin(c).

```cpp
template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));
```

Returns: std::end(c).

```cpp
template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());
```

Returns: c.rbegin().

```cpp
template <class C> auto rend(C& c) -> decltype(c.rend());
template <class C> auto rend(const C& c) -> decltype(c.rend());
```

Returns: c.rend().
Returns: c.rend().

```cpp
template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);
```

Returns: reverse_iterator<T*>(array + N).

```cpp
template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);
```

Returns: reverse_iterator<T*>(array).

```cpp
template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);
```

Returns: reverse_iterator<const E*>(il.end()).

```cpp
template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);
```

Returns: reverse_iterator<const E*>(il.begin()).

```cpp
template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));
```

Returns: std::rbegin(c).

```cpp
template <class C> auto crend(const C& c) -> decltype(std::rend(c));
```

Returns: std::rend(c).

### 24.11.1 begin

The name `begin` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::begin(E)` for some expression `E` is equivalent to:

1. **(1.1)** — `ranges::begin((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*]

2. **(1.2)** — Otherwise, `(E) + 0` if `E` has array type (3.9.2).

3. **(1.3)** — Otherwise, `(E).begin()` if its type `I` meets the syntactic requirements of `Iterator<I>()`. If `Iterator<I>()` is not satisfied, the program is ill-formed with no diagnostic required.

4. **(1.4)** — Otherwise, `begin(E)` if its type `I` meets the syntactic requirements of `Iterator<I>()` with overload resolution performed in a context that includes the declaration `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.

5. **(1.5)** — Otherwise, `ranges::begin(E)` is ill-formed.

### 24.11.2 end

The name `end` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::end(E)` for some expression `E` is equivalent to:

1. **(1.1)** — `ranges::end((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*]

2. **(1.2)** — Otherwise, `(E) + extent<T>::value` if `E` has array type (3.9.2) `T`. 

§ 24.11.2
2 \text{Remark: Whenever \texttt{ranges::end(E)} is a valid expression, the types of \texttt{ranges::end(E)} and \texttt{ranges::begin(E)} satisfy Sentinel.}

\textbf{24.11.4 \texttt{cend}} \quad \textbf{[iterator.range.cend]}

1 The name \texttt{cend} denotes a customization point object (17.5.2.1.5). The effect of the expression \texttt{ranges::cend(E)} for some expression \texttt{E} of type \texttt{T} is equivalent to \texttt{ranges::end((const T&)(E))}.

2 Use of \texttt{ranges::cend(E)} with rvalue \texttt{E} is deprecated. \textit{[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 \textit{[Note: Whenever \texttt{ranges::cend(E)} is a valid expression, the type of \texttt{ranges::cend(E)} satisfies Iterator. —end note]}

\textbf{24.11.5 \texttt{rbegin}} \quad \textbf{[iterator.range.rbegin]}

1 The name \texttt{rbegin} denotes a customization point object (17.5.2.1.5). The effect of the expression \texttt{ranges::rbegin(E)} for some expression \texttt{E} of type \texttt{T} is equivalent to:

\begin{enumerate}
\item \texttt{ranges::rbegin((const T&)(E))} if \texttt{E} is an rvalue of type \texttt{T}. This usage is deprecated. \textit{[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 rvalue. —end note]}
\item Otherwise, \texttt{make_reverse_iterator((E) + extent\texttt{T}::value)} if \texttt{E} has array type (3.9.2) \texttt{T}.
\item Otherwise, \texttt{(E).rbegin()} if its type \texttt{I} meets the syntactic requirements of \texttt{Iterator\texttt{I}()} . If \texttt{Iterator} is not satisfied, the program is ill-formed with no diagnostic required.
\item Otherwise, \texttt{make_reverse_iterator(ranges::end(E))} if both \texttt{ranges::begin(E)} and \texttt{ranges::end(E)} have the same type \texttt{I} which meets the syntactic requirements of \texttt{BidirectionalIterator\texttt{I}()} (24.2.18). If \texttt{BidirectionalIterator} is not satisfied, the program is ill-formed with no diagnostic required.
\item Otherwise, \texttt{ranges::rbegin(E)} is ill-formed.
\end{enumerate}

\textit{Remark: Whenever \texttt{ranges::rbegin(E)} is a valid expression, the type of \texttt{ranges::rbegin(E)} satisfies Iterator.}
24.11.6 rend

The name `rend` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::rend(E)` for some expression `E` is equivalent to:

1. `ranges::rend((const T&)(E))` if `E` is an rvalue 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 rvalue. — end note]

2. Otherwise, `make_reverse_iterator((E) + 0)` if `E` has array type (3.9.2).

3. Otherwise, `(E).rend()` if 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.

4. Otherwise, `make_reverse_iterator(ranges::begin(E))` if both `ranges::begin(E)` and `ranges::end(E)` have the same type `I` which meets the syntactic requirements of `BidirectionalIterator<I>()` (24.2.18). If `BidirectionalIterator` is not satisfied, the program is ill-formed with no diagnostic required.

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

Remark: Whenever `ranges::rend(E)` is a valid expression, the types of `ranges::rend(E)` and `ranges::rbegin(E)` satisfy `Sentinel`.

24.11.7 crbegin

The name `crbegin` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::crbegin(E)` for some expression `E` of type `T` is equivalent to `ranges::rbegin((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::crbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — end note]

[Note: Whenever `ranges::crbegin(E)` is a valid expression, the type of `ranges::crbegin(E)` satisfies `Iterator`. — end note]

24.11.8 crend

The name `crend` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::crend(E)` for some expression `E` of type `T` is equivalent to `ranges::rend((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]

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

24.12 Range primitives

Template `<_Auto C> auto size(const C& c) -> decltype(c.size());`

1

Returns: `c.size()`.

Template `<_Auto T, size_t N> constexpr size_t size(T (&array)[N]) noexcept;`

2

Returns: `N`.

Template `<_Auto E> constexpr size_t size(initializer_list<E> il) noexcept;`

3

Returns: `il.size()`.
Range<R>::template <Range R>
DifferenceType_difference_type_t<IteratorType_iterator_t<R>> distance(R& r);

Returns: ranges::vi::distance(ranges::begin(r), ranges::end(r))

SizedRange<R>::template <SizedRange R>
DifferenceType_difference_type_t<IteratorType_iterator_t<R>> distance(R& r);

Returns: ranges::size(r)

24.12.1 size

The name `size` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::size(E)` for some expression `E` with type `T` is equivalent to:

\begin{enumerate}
\item extent<T>::value if `T` is an array type (3.9.2).
\item Otherwise, `(const T&)E).size()` if its type `I` satisfies `Integral<const I>()`.
\item Otherwise, `size((const T&)E)` with overload resolution performed in a context that includes the declaration `void size(const auto&) = delete;` and does not include a declaration of `ranges::size`.
\item Otherwise, `ranges::size(E)` is ill-formed.
\end{enumerate}

\begin{enumerate}
\item `[Note: Whenever `ranges::size(E)` is a valid expression, the type of `ranges::size(E)` satisfies `Integral`. —end note]`
\end{enumerate}

24.12.2 empty

The name `empty` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::empty(E)` for some expression `E` is equivalent to:

\begin{enumerate}
\item bool((E).empty()) if it is valid.
\item Otherwise, `ranges::size(E) != 0` if it is valid.
\item Otherwise, `bool(ranges::begin(E) != ranges::end(E))`, except that `E` is only evaluated once.
\item Otherwise, `ranges::empty(E)` is ill-formed.
\end{enumerate}

Remark: Whenever `ranges::empty(E)` is a valid expression, it has type `bool`.

24.12.3 data

The name `data` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::data(E)` for some expression `E` is equivalent to:

\begin{enumerate}
\item `ranges::data((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]
\item Otherwise, `(E) + 0` if `E` has array type (3.9.2).
\item Otherwise, `E.data()` if it has pointer to object type.
\item Otherwise, `ranges::data(E)` is ill-formed.
\end{enumerate}

Remark: Whenever `ranges::data(E)` is a valid expression, it has type `std::pointer_to_object_type`.

§ 24.12.3
24.12.4 cdata

The name `cdata` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::cdata(E)` for some expression `E` of type `T` is equivalent to `ranges::data((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]

[Note: Whenever `ranges::cdata(E)` is a valid expression, it has pointer to object type. — end note]
25 Algorithms library

25.1 General

This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 23) and other sequences.

The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 7.

Table 7 — Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.2 Non-modifying sequence operations</td>
<td>&lt;experimental/ranges_v1/algorithm&gt;</td>
</tr>
<tr>
<td>25.3 Mutating sequence operations</td>
<td>&lt;experimental/ranges_v1/algorithm&gt;</td>
</tr>
<tr>
<td>25.4 Sorting and related operations</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>25.5 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

Header <experimental/ranges_v1/algorithm> synopsis

```cpp
#include <initializer_list>

namespace std {
    namespace experimental {
        namespace ranges_v1 {
            namespace v1 {
                namespace tag {
                    // 25.1, tag specifiers (See 20.15.2):
                    struct input;
                    struct input1;
                    struct input2;
                    struct output;
                    struct output1;
                    struct output2;
                    struct fun;
                    struct min;
                    struct max;
                    struct begin;
                    struct end;
                }

                // 25.2, non-modifying sequence operations:
                template<InputIterator I, Sentinel<I> S, class Proj = identity,
                        IndirectCallablePredicate<Projected<I, Proj>> Pred>
                bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

                template<InputRange Rng, class Proj = identity,
                        IndirectCallablePredicate<Projected<IteratorTypeIterator_t<Rng>, Proj>> Pred>
                bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});

                template<InputIterator I, Sentinel<I> S, class Proj = identity,
                        IndirectCallablePredicate<Projected<I, Proj>> Pred>
                bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

                template<InputRange Rng, class Proj = identity,
                        IndirectCallablePredicate<Projected<IteratorTypeIterator_t<Rng>, Proj>> Pred>
                bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});

                template<InputIterator I, Sentinel<I> S, class Proj = identity,
                        IndirectCallablePredicate<Projected<I, Proj>> Pred>
                bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

                template<InputRange Rng, class Proj = identity,
                        IndirectCallablePredicate<Projected<IteratorTypeIterator_t<Rng>, Proj>> Pred>
                bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
            }
        }
    }
}
```
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred
bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<Projected<I, Proj>> Pred>
bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallable<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,
IndirectCallable<Projected<IteratorType iterator_t<Rng>, Proj>> Fun>
tagged_pair<tag::in(IteratorType 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 IndirectCallableRelation<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 IndirectCallableRelation<equal_to<>, Projected<IteratorType iterator_t<Rng>, Proj>, const T*>()
IteratorType safe_iterator_t<Rng>
find(Rng&& rng, const T& value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<Projected<I, Proj>> Pred>
I find_if(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
IteratorType safe_iterator_t<Rng>
find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<Projected<I, Proj>> Pred>
I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
IteratorType safe_iterator_t<Rng>
find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});

template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
Sentinel<I2> S2, class Proj = identity,
IndirectCallableRelation<I2, Projected<I1, Proj>, Pred = equal_to<>> II
find_end(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{}, Proj proj = Proj{});
template<ForwardRange Rng1, ForwardRange Rng2, class Proj = identity, 
    IndirectCallableRelation<IteratorType iterator_t<Rng2>, 
    Projected<IteratorType iterator_t<Rng>, Proj>> Pred = equal_to<>>
IteratorType 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,
    IndirectCallablePredicate<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,
    IndirectCallablePredicate<Projected<I1, Proj1>, Projected<I2, Proj2>> Pred = equal_to<>>
IteratorType safe_iterator_t<Rng1>
find_first_of(Rng1& rng1, Rng2&& rng2,
    Pred pred = Pred{},
    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Pred = equal_to<>>
I adjacent_find(I first, S last, Pred pred = Pred{},
    Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType iterator_t<Rng>, Proj>> Pred = equal_to<>>
IteratorType safe_iterator_t<Rng>
adjacent_find(Rng& rng, Pred pred = Pred{}, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectCallableRelation<equal_to,, Projected<I, Proj>>, const T*>(
    DifferenceType difference_type_t<I>
    count(I first, S last, const T& value, Proj proj = Proj{});

template<InputRange Rng, class T, class Proj = identity>
    requires IndirectCallableRelation<equal_to,, Projected<IteratorType iterator_t<Rng>, Proj>>, const T*>(
    DifferenceType difference_type_t<IteratorType iterator_t<Rng>>
    count(Rng& rng, const T& value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
DifferenceType difference_type_t<I>
count_if(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
DifferenceType difference_type_t<IteratorType iterator_t<Rng>>
count_if(Rng& rng, Pred pred, Proj proj = Proj{});

// D.10 (deprecated):
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2, 
  class Proj1 = identity, class Proj2 = identity, 
  IndirectCallablePredicate<Projected<I1, Proj1>, Projected<I2, Proj2>> Pred = equal_to<>>
tagged_pair<tag::in1(I1), tag::in2(I2)>
mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{}, 
  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<InputRange Rng1, WeakInputIterator I2, 
  class Proj1 = identity, class Proj2 = identity, 
  IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng1>, Proj1>, 
  Projected<IteratorType iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
tagged_pair<tag::in1(iterator_t<Rng1)>, tag::in2(I2)>
mismatch(Rng1&& rng1, I2 first2, Pred pred = Pred{}, 
  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, 
  class Proj1 = identity, class Proj2 = identity, 
  IndirectCallablePredicate<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, 
  IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng1>, Proj1>, 
  Projected<IteratorType iterator_t<Rng2>, Proj2>>Pred = equal_to<>>
tagged_pair<tag::in1(iterator_t<Rng1)>, tag::in2(iterator_t<Rng2)>
mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, 
  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2, 
  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, Pred pred = Pred{}, 
  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<InputRange Rng1, WeakInputIterator I2, class Pred = equal_to<>>, 
  class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<IteratorType iterator_t<Rng1>, I2, Pred, Proj1, Proj2>,bool equal(Rng1&& rng1, I2 first2, 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<>>, 

§ 25.1
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>, Pred, Proj1, Proj2>
bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
          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,
                   Pred pred = Pred{},
                   Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<ForwardRange Rng1, ForwardIterator I2, class Pred = equal_to<>,
          class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<IteratorType iterator_t<Rng1>, I2, Pred, Proj1, Proj2>
bool is_permutation(Rng1&& rng1, I2 first2, 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<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>, Pred, Proj1, Proj2>
bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                   Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
          Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
          class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
I
I1
  search(I1 first1, S1 last1, I2 first2, S2 last2,
         Pred pred = Pred{},
         Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
          class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>, Pred, Proj1, Proj2>
  IteratorType safe_iterator_t<Rng1>
  search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
         Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<ForwardIterator I, Sentinel<I> S, class T,
          class Pred = equal_to<>, class Proj = identity>
requires IndirectlyComparable<I, const T*, Pred, Proj>
I
  search_n(I first, S last, DifferenceType difference_type_t<I> count,
const T& value, Pred pred = Pred{}, Proj proj = Proj{});

template<ForwardRange Rng, class T, class Pred = equal_to<>, class Proj = identity>
requires IndirectlyComparable<IteratorType iterator_t<Rng>, const T*, Pred, Proj>()
IteratorType safe_iterator_t<Rng>
safe_iterator_t<Rng, DifferenceType difference_type_t<IteratorType iterator_t<Rng>> count,
const T& value, Pred pred = Pred{}, Proj proj = Proj{});

// 25.3, modifying sequence operations:
// 25.3.1, 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<IteratorType iterator_t<Rng>, O>,
tagged_pair<tag::in((IteratorType) safe_iterator_t<Rng>), tag::out(O)>
copy(Rng& rng, O result);

template<WeakInputIterator I, WeaklyIncrementable O>
requires IndirectlyCopyable<I, O>,
tagged_pair<tag::in(I), tag::out(O)>
copy_n(I first, iterator_distance_t<I> n, O result);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
IndirectCallablePredicate<Projected<I, Proj>> Pred>
requires IndirectlyCopyable<I, O>,
tagged_pair<tag::in(I), tag::out(O)>
copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template<InputRange Rng, WeaklyIncrementable O, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
requires IndirectlyCopyable<IteratorType iterator_t<Rng>, O>,
tagged_pair<tag::in((IteratorType) safe_iterator_t<Rng>), tag::out(O)>
copy_if(Rng& rng, O result, Pred pred, Proj proj = Proj{});

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
requires IndirectlyCopyable<I1, I2>,
tagged_pair<tag::in(I1), tag::out(I2)>
copy_backward(I1 first, S1 last, I2 result);

template<BidirectionalRange Rng, BidirectionalIterator I>
requires IndirectlyCopyable<IteratorType iterator_t<Rng>, I>,
tagged_pair<tag::in((IteratorType) safe_iterator_t<Rng>), tag::out(I)>
copy_backward(Rng& rng, I result);

// 25.3.2, 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, O result);
template<InputRange Rng, WeaklyIncrementable O>
 requires IndirectlyMovable<IteratorType iterator_t<Rng>, O>()
tagged_pair<tag::in(IteratorType safe_iterator_t<Rng>), tag::out(O)>
 move(Rng& rng, O result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
 requires IndirectlyMovable<I1, I2>()
tagged_pair<tag::in(I1), tag::in2Out(I2)>
 move_backward(I1 first, S1 last, I2 result);

template<BidirectionalRange Rng, BidirectionalIterator I>
 requires IndirectlyMovable<IteratorType iterator_t<Rng>, I>()
tagged_pair<tag::in(IteratorType safe_iterator_t<Rng>), tag::in2Out(I)>
 move_backward(Rng& rng, I result);

// 25.3.3, swap, D.10 (deprecated):
template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
 requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in1(I1), tag::in2(I2)>
 swap_ranges(I1 first1, S1 last1, I2 first2);

// D.10 (deprecated):
template<ForwardRange Rng, ForwardIterator I>
 requires IndirectlySwappable<IteratorType iterator_t<Rng>, I>()
tagged_pair<tag::in1(IteratorType safe_iterator_t<Rng>), tag::in2(I)>
 swap_ranges(Rng& rng1, I first2);

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<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>>()
tagged_pair<tag::in1(IteratorType safe_iterator_t<Rng1>), tag::in2(IteratorType safe_iterator_t<Rng2>)>
 swap_ranges(Rng1& rng1, Rng2& rng2);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class F, class Proj = identity>
 requires IndirectlyCallable<Projected<I, Proj>>, WeakOutputIterator<IndirectCallableResultType<F, Projected<I, Proj>>> O
 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, class F, class Proj = identity>
 requires IndirectlyCallable<Projected<IteratorType iterator_t<Rng>>, Proj>, WeakOutputIterator<IndirectCallableResultType<F, Projected<IteratorType iterator_t<Rng>>, Proj>>> O
 tagged_pair<tag::in(IteratorType safe_iterator_t<Rng>), tag::out(O)>
 transform(Rng& rng, O result, F op, Proj proj = Proj());

// D.10 (deprecated):
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2, WeaklyIncrementable O, class F, class Proj = identity>
 requires IndirectlyCallable<Projected<InputIterator I, WeakInputIterator I2>, Proj>> O
 tagged_pair<tag::in(I), tag::out(O)>
 transform(I first, S last, O result, F op, Proj proj = Proj());

§ 25.1
class F, class Proj1 = identity, class Proj2 = identity,
IndirectCallable<Projected<I1, Proj1>, Projected<I2, Proj2>> F,
WeakOutputIterator<IndirectCallableResultType<F, Projected<I1, Proj1>>, Projected<I2, Proj2>> O
requires Writable<O, IndirectCallableResultType<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, O result,
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<InputRange Rng, WeakInputIterator I, WeaklyIncrementable O, class F,
class Proj1 = identity, class Proj2 = identity,
IndirectCallable<Projected<IteratorType<Rng>, Proj1>, Projected<I, Proj2>> F,
WeakOutputIterator<IndirectCallableResultType<F, Projected<IteratorType<Rng>, Proj1>>, Projected<I, Proj2>> O
requires Writable<O, IndirectCallableResultType<F, Projected<IteratorType<Rng>, Proj1>>, Projected<I, Proj2>>()
tagged_tuple<tag::in1(IteratorTypesafe_iterator_t<Rng>), tag::in2(I), tag::out(O)>
transform(Rng& rng1, I first2, O result,
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
WeaklyIncrementable O, class F, class Proj1 = identity, class Proj2 = identity,
IndirectCallable<Projected<I1, Proj1>, Projected<I2, Proj2>> F,
WeakOutputIterator<IndirectCallableResultType<F, Projected<I1, Proj1>>, Projected<I2, Proj2>> O
requires Writable<O, IndirectCallableResultType<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, class F,
class Proj1 = identity, class Proj2 = identity,
IndirectCallable<Projected<IteratorType<Rng1>, Proj1>, Projected<IteratorType<Rng2>, Proj2>> F,
WeakOutputIterator<IndirectCallableResultType<F, Projected<IteratorType<Rng1>, Proj1>>, Projected<IteratorType<Rng2>, Proj2>> O
requires Writable<O, IndirectCallableResultType<F, Projected<IteratorType<Rng1>, Proj1>>, Projected<IteratorType<Rng2>, Proj2>>()
tagged_tuple<tag::in1(IteratorTypesafe_iterator_t<Rng1>), tag::in2(IteratorTypesafe_iterator_t<Rng2>), tag::out(O)>
transform(Rng1& rng1, Rng2& rng2, O result,
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

[Editor’s note: REVIEW: In the Palo Alto proposal, replace requires only InputIterators. In C++14, it requires Forward.]

template<ForwardIterator I, Sentinel<I> S, class T1, Semiregular class T2, class Proj = identity>
requires Writable<I, T2> & &

§ 25.1 123
IndirectCallableRelation<equal_to<>, P(projected<I, Proj>>, const T1*>>

I replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj());

template<ForwardRange Rng, class T1, Semiregular class T2, class Proj = identity>
requires Writable<IteratorType_safe_iterator_t<Rng>, T1 &>
IndirectCallableRelation<equal_to<>, P(projected<IteratorType_safe_iterator_t<Rng>, Proj>>, const T1*>>
IteratorType_safe_iterator_t<Rng>
replace(Rng& rng, const T1& old_value, const T2& new_value, Proj proj = Proj());

template<ForwardIterator I, Sentinel<I> S, Semiregular class T, class Proj = identity, IndirectCallablePredicate<P(projected<I, Proj>>, Pred>>
requires Writable<I, T &>
I replace_if(I first, S last, Pred pred, const T1& new_value, Proj proj = Proj());

template<ForwardRange Rng, Semiregular class T, class Proj = identity, IndirectCallablePredicate<P(projected<IteratorType_safe_iterator_t<Rng>, Proj>>, T &>>
IteratorType_safe_iterator_t<Rng>
replace_if(Rng& rng, Pred pred, const T1& new_value, Proj proj = Proj());

template<InputIterator I, Sentinel<I> S, class T1, Semiregular class T2, WeakOutputIterator<T2> O, class Proj = identity>
requires IndirectlyCopyable<I, O &>
IndirectCallableRelation<equal_to<>, P(projected<I, Proj>>, const T1*>>
tagged_pair<tag::in(I), tag::out(O)>
replace_copy(I first, S last, 0 result, const T1& old_value, const T2& new_value, Proj proj = Proj());

template<InputRange Rng, class T1, Semiregular class T2, WeakOutputIterator<T2> O, class Proj = identity, IndirectCallablePredicate<P(projected<IteratorType_safe_iterator_t<Rng>, O &>>, T &>>
tagged_pair<tag::in(IteratorType_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, Semiregular class T, WeakOutputIterator<T2> O, class Proj = identity, IndirectCallablePredicate<P(projected<I, Proj>>, Pred>>
requires IndirectlyCopyable<I, O &>
tagged_pair<tag::in(I), tag::out(O)>
replace_copy_if(I first, S last, 0 result, Pred pred, const T1& new_value, Proj proj = Proj());

template<InputRange Rng, Semiregular class T, WeakOutputIterator<T2> O, class Proj = identity, IndirectCallablePredicate<P(projected<IteratorType_safe_iterator_t<Rng>, O &>>, Pred>>
tagged_pair<tag::in(IteratorType_safe_iterator_t<Rng>), tag::out(O)>
replace_copy_if(Rng& rng, Pred pred, const T1& new_value, Proj proj = Proj());

[Editor's note: REVIEW: N3351 only requires WeaklyIncrementable for fill and generate]
template<typename class T, typename OutputIterator<T> O, typename Sentinel<O> S>
  O fill(O first, S last, const T& value);

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

template<typename class T, typename WeakOutputIterator<T> O>
  O fill_n(O first, DistanceType n, const T& value);

template<typename Function F, typename OutputIterator<ResultType<F> result_of_t<F&()>> O, typename Sentinel<O> S>
  O generate(O first, S last, F gen);

template<typename Function F, typename OutputRange<ResultType<F> result_of_t<F&()>> Rng>
  IteratorType safe_iterator_t<Rng>
    generate(Rng& rng, F gen);

template<typename Function F, typename WeakOutputIterator<ResultType<F> result_of_t<F&()>> O>
  O generate_n(O first, DistanceType n, F gen);

template<typename ForwardIterator I, typename Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I>() &&
    IndirectCallableRelation<!equal_to<>, projected<I, Proj>, const T*>(&I)
  I remove(I first, S last, const T& value, Proj proj = Proj());

template<typename ForwardRange Rng, class T, class Proj = identity>
  requires Permutable<IteratorType iterator_t<Rng>>() &&
    IndirectCallableRelation<!equal_to<>, projected<IteratorType iterator_t<Rng>, Proj>, const T*>(&I)
  remove(Rng& rng, const T& value, Proj proj = Proj());

template<typename ForwardIterator I, typename Sentinel<I> S, class Proj = identity, 
          IndirectCallablePredicate<projected<I, Proj>> Pred>
  requires Permutable<I>()
  I remove_if(I first, S last, Pred pred, Proj proj = Proj());

template<typename ForwardRange Rng, class Proj = identity, 
          IndirectCallablePredicate<projected<IteratorType iterator_t<Rng>, Proj>> Pred>
  requires Permutable<IteratorType iterator_t<Rng>>()
  remove_if(Rng& rng, Pred pred, Proj proj = Proj());

template<typename InputIterator I, typename Sentinel<I> S, class T, class Proj = identity>
  requires IndirectlyCopyable<I, O>() &&
    IndirectCallableRelation<!equal_to<>, projected<I, Proj>, const T*>(&I)
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy(I first, S last, O result, const T& value, Proj proj = Proj());

template<typename InputRange Rng, class Proj = identity>
  requires IndirectlyCopyable<IteratorType iterator_t<Rng>, O>() &&
    IndirectCallableRelation<!equal_to<>, projected<IteratorType iterator_t<Rng>, Proj>, const T*>(&I)
  tagged_pair<tag::in(IteratorType 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, IndirectCallablePredicate<
   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,  
   IndirectCallablePredicate<
   projected<IteratorType iterator_t<Rng>, Proj>> Pred>
   requires IndirectlyCopyable<
   IteratorType iterator_t<Rng>, O>()
   tagged_pair<tag::in(IteratorType 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,  
   IndirectCallableRelation<
   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,  
   IndirectCallableRelation<
   projected<IteratorType iterator_t<Rng>, Proj>> R = equal_to<>>
   requires Permutable<
   IteratorType iterator_t<Rng>()>
   IteratorType safe_iterator_t<Rng>
   unique(Rng& rng, R comp = R{}, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,  
   class Proj = identity, IndirectCallableRelation<
   projected<I, Proj>> R = equal_to<>>
   requires IndirectlyCopyable<I, O>() && (ForwardIterator<I>() | | 
   ForwardIterator<O>() | | Copyable<ValueType value_type_t<I>>())
   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,  
   IndirectCallableRelation<
   projected<IteratorType iterator_t<Rng>, Proj>> R = equal_to<>>
   requires IndirectlyCopyable<
   IteratorType iterator_t<Rng>, O>() && (ForwardIterator<
   IteratorType iterator_t<Rng>() | | ForwardIterator<O>() | | 
   Copyable<ValueType value_type_t<
   IteratorType iterator_t<Rng>()>>())
   tagged_pair<tag::in(IteratorType 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<
   IteratorType iterator_t<Rng>>()
   IteratorType 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<
   IteratorType iterator_t<Rng>, O>()
   tagged_pair<tag::in(IteratorType safe_iterator_t<Rng>), tag::out(O)>
   reverse_copy(Rng& rng, O result);
reverse_copy(Rng& rng, 0 result);

[Editor’s note: Could return a range instead of a pair. See Future Work annex (C.2).]

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<IteratorType iterator_t<Rng>>()
    tagged_pair<tag::begin(IteratorType safe_iterator_t<Rng>),
    tag::end(IteratorType safe_iterator_t<Rng>)>
    rotate(Rng& rng, IteratorType iterator_t<Rng> middle);

template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>()
    tagged_pair<tag::begin(I), tag::out(O)>
    rotate_copy(I first, I middle, S last, 0 result);

template<ForwardRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<IteratorType iterator_t<Rng>, O>()
    tagged_pair<tag::begin(IteratorType safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng& rng, IteratorType iterator_t<Rng> middle, 0 result);

// 25.3.12, shuffle:

template<RandomAccessIterator I, Sentinel<I> S, class Gen>
    requires Permutable<I>() && ConvertibleTo<ResultType<Gen> result_of_t<Gen&()>, DifferenceType difference_type_t<I>>()
    UniformRandomNumberGenerator<remove_reference_t<Gen>>()
    I shuffle(I first, S last, Gen&& g);

template<RandomAccessRange Rng, class Gen>
    requires Permutable<I>() && ConvertibleTo<ResultType<Gen> result_of_t<Gen&()>, DifferenceType difference_type_t<I>>()
    UniformRandomNumberGenerator<remove_reference_t<Gen>>
    IteratorType safe_iterator_t<Rng>
    shuffle(Rng& rng, Gen&& g);

// 25.3.13, partitions:

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<Projected<I, Proj>> Pred>
    bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
    bool is_partitioned(Rng& rng, Pred pred, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
    requires Permutable<I>()
    I partition(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
requires Permutable<IteratorType, iterator_t<Rng>>;
IteratorType safe_iterator_t<Rng>
partition(Rng& rng, Pred pred, Proj proj = Proj{});

template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
  IndirectCallablePredicate<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,
  IndirectCallablePredicate<Projected<IteratorType, iterator_t<Rng>>, Proj>> Pred>
requires Permutable<IteratorType, iterator_t<Rng>>()
IteratorType safe_iterator_t<Rng>
stable_partition(Rng& rng, Pred pred, Proj proj = Proj{});

[Editor's note: A new algorithm, needed by stable_partition.]

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
  class Proj = identity, IndirectCallablePredicate<Projected<I, Proj>> Pred>
requires IndirectlyCopyable<I, O1>() && IndirectlyCopyable<I, O2>()
tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
  partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
    Proj proj = Proj{});

template<InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
  class Proj = identity,
  IndirectCallablePredicate<Projected<IteratorType, iterator_t<Rng>>, Proj>> Pred>
requires IndirectlyCopyable<IteratorType, iterator_t<Rng>(), O1>() &&
  IndirectlyCopyable<IteratorType, iterator_t<Rng>(), O2>()
tagged_tuple<tag::in(IteratorType safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
  partition_copy(Rng& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
  class Proj = identity,
  IndirectCallablePredicate<Projected<I, Proj>> Pred>
requires IndirectlyMovable<I, O1>() && IndirectlyMovable<I, O2>()
tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
  partition_move(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,
  IndirectCallablePredicate<Projected<IteratorType, iterator_t<Rng>>, Proj>> Pred>
requires IndirectlyMovable<IteratorType, iterator_t<Rng>(), O1>() &&
  IndirectlyMovable<IteratorType, iterator_t<Rng>(), O2>()
tagged_tuple<tag::in(IteratorType safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
  partition_move(Rng& rng, O1 out_true, O2 out_false, Pred pred,
    Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
  IndirectCallablePredicate<Projected<I, Proj>> Pred>
I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred
IteratorType safe_iterator_t<Rng>
partition_point(Rng& rng, Pred pred, Proj proj = Proj{});

// 25.4, sorting and related operations:
// 25.4.1, 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<IteratorType iterator_t<Rng>, Comp, Proj>()
    IteratorType 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<IteratorType iterator_t<Rng>, Comp, Proj>()
    IteratorType 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<IteratorType iterator_t<Rng>, Comp, Proj>()
    IteratorType safe_iterator_t<Rng>
    partial_sort(Rng& rng, iterator_t<Rng> middle, Comp comp = Comp{}, Proj proj = Proj{});

template<InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
    class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
    requires IndirectlyCopyable<I1, I2> & & Sortable<I2, Comp, Proj2>() & &
    IndirectCallableStrictWeakOrder<Comp, Projected<I1, Proj1>, Projected<I2, Proj2>>()
    I2 partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, RandomAccessRange Rng2, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyCopyable<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>> & &
    Sortable<IteratorType iterator_t<Rng2>, Comp, Proj2>() & &
    IndirectCallableStrictWeakOrder<Comp, Projected<IteratorType iterator_t<Rng1>, Proj1>,
    Projected<IteratorType iterator_t<Rng2>, Proj2>>()
    IteratorType 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,
          IndirectCallableStrictWeakOrder<const projected<I, Proj>>, Comp = less<>>
bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
          IndirectCallableStrictWeakOrder<const IteratorType_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,
          IndirectCallableStrictWeakOrder<const projected<I, Proj>>, Comp = less<>>
I
is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
          IndirectCallableStrictWeakOrder<const IteratorType_iterator_t<Rng>, Proj>>, Comp = less<>>
I
is_sorted_until(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
          class Proj = identity>
I
nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessRange Rng, class Comp = less<>,
          class Proj = identity>
IteratorType_safe_iterator_t<Rng>
nth_element(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

// 25.4.3, binary search:

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

template<ForwardRange Rng, TotallyOrderedClass T, class Proj = identity,
          IndirectCallableStrictWeakOrder<const IteratorType_iterator_t<Rng>, Proj>>, Comp = less<>>
IteratorType_safe_iterator_t<Rng>
lower_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, TotallyOrderedClass T, class Proj = identity,
          IndirectCallableStrictWeakOrder<const IteratorType_iterator_t<I>, Proj>>, Comp = less<>>
I
upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, TotallyOrderedClass T, class Proj = identity,
          IndirectCallableStrictWeakOrder<const IteratorType_iterator_t<Rng>, Proj>>, Comp = less<>>
IteratorType_safe_iterator_t<Rng>
upper_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

[Editor's note: This could return a range instead of a pair. See the Future Work annex (C.2).]

§ 25.1
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, TotallyOrdered class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, Projected<IteratorType iterator_t<Rng>>, Proj>>
tagged_pair<tag::begin(iterator_t<Rng>), tag::end(iterator_t<Rng>)>
    equal_range(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, TotallyOrdered class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, Projected<IteratorType iterator_t<Rng>>, Proj>>
bool
    binary_search(IteratorType safe_iterator_t<Rng>, const T& value, Comp comp = Comp{},
                    Proj proj = Proj{});

template<ForwardRange Rng, TotallyOrdered class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, Projected<IteratorType iterator_t<Rng>>, Proj>>
binary_search(Rng& rng, const T& value, Comp comp = Comp{},
                   Proj proj = Proj{});

    // 25.4.4, merge:

[Editor's note: REVIEW: Why does the Palo Alto TR require Incrementable instead of WeaklyIncrementable?]
template<InputRange Rng1, InputRange Rng2, Incrementable O, class Comp = less<>,
class Proj1 = identity, class Proj2 = identity>
requires MergeMovable<
  IteratorType iterator_t<Rng1>,
  IteratorType iterator_t<Rng2>,
  O, Comp, Proj1, Proj2>

  tagged_tuple<
    ::in1(IteratorType safe_iterator_t<Rng1>),
    ::in2(IteratorType safe_iterator_t<Rng2>),
    ::out(O)>
merge_move(Rng1&& rng1, Rng2&& rng2, O result,
  Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
class Proj = identity>
requires Sortable<I, Comp, Proj>

  I inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<InputRange Rng1, InputRange Rng2, class Proj1 = identity,
class Proj2 = identity,
IndirectCallableStrictWeakOrder<
  projected<IteratorType iterator_t<Rng1>, Proj1>,
  projected<IteratorType iterator_t<Rng2>, Proj2>> Comp = less<>>
bool includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

§ 25.1 132
requires Mergeable<I1, I2, 0, Comp, Proj1, Proj2>()
set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, 0 result,
   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable 0,
   class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<typename iterator_t<Rng1>, iterator_t<Rng2>, 0, Comp, Proj1, Proj2>()
set_intersection(Rng1 & rng1, Rng2 & rng2, 0 result,
   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
   WeaklyIncrementable 0, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<I1, I2, 0, Comp, Proj1, Proj2>()
tagged_pair<tag::in1(I1), tag::out(O)>
set_difference(I1 first1, S1 last1, I2 first2, S2 last2, 0 result,
   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable 0,
   class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<typename iterator_t<Rng1>, iterator_t<Rng2>, 0, Comp, Proj1, Proj2>()
tagged_pair<tag::in1(iterator_t<Rng1>), tag::out(O)>
set_difference(Rng1 & rng1, Rng2 & rng2, 0 result,
   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 25.4.6, 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<typename iterator_t<Rng>, Comp, Proj>()
iterator_t<Rng> push_heap(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

§ 25.1
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<IteratorType_iterator_t<Rng>, Comp, Proj>()
IteratorType_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<IteratorType_iterator_t<Rng>, Comp, Proj>()
IteratorType_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<IteratorType_iterator_t<Rng>, Comp, Proj>()
IteratorType_safe_iterator_t<Rng>
sort_heap(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<I, Proj>> Comp = less<>>
bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessRange Rng, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<IteratorType_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,
IndirectCallableStrictWeakOrder<P projected<I, Proj>> Comp = less<>>
bool
is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessRange Rng, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<IteratorType_iterator_t<Rng>, Proj>> Comp = less<>>
IteratorType_safe_iterator_t<Rng>
is_heap_until(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

// 25.4.7, minimum and maximum:
template<TotallyOrdered T>
constexpr const T& min(const T& a, const T& b);

template<class T, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<const T*, Proj>> Comp = less<>>
requires StrictWeakOrder<FunctionType<Comp>, T>
constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

§ 25.1
Editor's note: REVIEW: The Palo Alto report returns by const reference here but the current document returns by value.

```c
template<TotallyOrdered T>
    requires Semiregular<T>
    constexpr T min(initializer_list<T> t);

template<InputRange Rng>
    requires TotallyOrdered<ValueType<IteratorType<Rng>>>()
    && Semiregular<ValueType<IteratorType<Rng>>>
    ValueType<IteratorType<Rng>>
    min(Rng&& rng);

template<SemiregularCopyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<Pprojected<const T*, Proj>> Comp = less>[
    requires StrictWeakOrder<FunctionType<Comp>, T>
    ]
    constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<Pprojected<IteratorType<Rng>, Proj>> Comp = less>[
    requires SemiregularCopyable<ValueType<ValueType_t<IteratorType<Rng>>>()
    ]
    ValueType<ValueType_t<IteratorType<Rng>>>
    min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<TotallyOrdered T>
    constexpr const T& max(const T& a, const T& b);

template<class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<Pprojected<const T*>, Proj>> Comp = less>[
    requires SemiregularCopyable<ValueType<ValueType_t<IteratorType>Rng>()
    ]
    ValueType<ValueType_t<IteratorType<Rng>>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<TotallyOrdered T>
    requires Semiregular<T>
    constexpr T max(initializer_list<T> t);

template<InputRange Rng>
    requires TotallyOrdered<ValueType<IteratorType<Rng>>>()
    && Semiregular<ValueType<IteratorType<Rng>>>
    ValueType<IteratorType<Rng>>
    max(Rng&& rng);

template<SemiregularCopyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<Pprojected<const T*, Proj>> Comp = less>[
    requires StrictWeakOrder<FunctionType<Comp>, T>
    ]
    constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<Pprojected<IteratorType<Rng>, Proj>> Comp = less>[
    requires SemiregularCopyable<ValueType<ValueType_t<IteratorType>Rng>()
    ]
    ValueType<ValueType_t<IteratorType<Rng>>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<TotallyOrdered T>
    constexpr tagged_pair<tag::min(const T&), tag::max(const T&)> minmax(const T& a, const T& b);
```

§ 25.1 135
template<class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less>
requires StrictWeakOrder<FunctionType<Comp>, T>
constexpr tagged_pair<tag::min(const T&), tag::max(const T&)> minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template<TotallyOrdered T>
requires Semiregular T
constexpr tagged_pair<tag::min(T), tag::max(T)> minmax(initializer_list<T> t);

template<InputRange Rng>
requires TotallyOrdered<IteratorType<Rng>> &
    Semiregular<ValueType<IteratorType<Rng>>>
    tagged_pair<tag::min(ValueType<IteratorType<Rng>>), tag::max(ValueType<IteratorType<Rng>>)> minmax(Rng&& rng);

template<Semiregular Copyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less>
requires StrictWeakOrder<FunctionType<Comp>, T>
constexpr tagged_pair<tag::min(T), tag::max(T)> minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<IteratorType<Rng>, Proj>> Comp = less>
requires Semiregular Copyable<ValueType<IteratorType<Rng>>, Proj>
tagged_pair<tag::min(ValueType<IteratorType<Rng>>, value_type_t<IteratorType<Rng>>), tag::max(ValueType<IteratorType<Rng>>, value_type_t<IteratorType<Rng>>)> minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<I, Proj>> Comp = less>
I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<IteratorType<Rng>, Proj>> Comp = less>
IteratorType safe_iterator_t<Rng> min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<I, Proj>> Comp = less>
I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<IteratorType<Rng>, Proj>> Comp = less>
IteratorType safe_iterator_t<Rng> max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<I, Proj>> Comp = less>
tagged_pair<tag::min(I), tag::max(I)> minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
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.

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.

[Editor’s note: The following paragraphs are removed because they are redundant; these requirements are now enforced in code by the requires clauses.]

Throughout this Clause, the names of template parameters are used to express type requirements. If an algorithm’s template parameter is InputIterator, InputIterator1, or InputIterator2, the actual template
argument shall satisfy the requirements of an input iterator (24.2.14). If an algorithm’s template parameter is `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the actual template argument shall satisfy the requirements of an output iterator (24.2.16). If an algorithm’s template parameter is `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the actual template argument shall satisfy the requirements of a forward iterator (24.2.17). If an algorithm’s template parameter is `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the actual template argument shall satisfy the requirements of a bidirectional iterator (24.2.18). If an algorithm’s template parameter is `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the actual template argument shall satisfy the requirements of a random-access iterator (24.2.19).

6 If an algorithm’s Effects section says that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (24.2). [Note: This requirement does not affect arguments that are declared as `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable. —end note]

7 Both in-place and copying versions are provided for certain algorithms.3 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`).

8 The `Predicate` parameter is used whenever an algorithm expects a function object (20.9) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (Clause 4). The function object `pred` shall not apply any non-constant function through the dereferenced iterator.

9 The `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (Clause 4). `BinaryPredicate` always takes the first iterator’s `value_type` as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (Clause 4). `binary_pred` shall not apply any non-constant function through the dereferenced iterators.

10 [Note: Projections and predicates are typically used as follows:

```cpp
auto&& proj_ = callable__as_function(proj);  // see 24.3.2
auto&& pred_ = callable__as_function(pred);
if(pred_(proj_(*first))) // ...
```

—end note]

11 [Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.9.3), or some equivalent solution. —end note]

12 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either true or false in boolean contexts.

13 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

---

3) 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`. 

§ 25.1
X tmp = a;
advance(tmp, n);
return tmp;

and that of \( b-a \) is the same as of

\[
\text{return distance}(a, b);
\]

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:

\[
I \text{ tmp } = \text{first};
\text{while}(\text{tmp }!= \text{last})
++\text{tmp} ;
\text{return tmp};
\]

Overloads of algorithms that take Range arguments (24.10.2.2) 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.

[Editor’s note: Before [alg.nonmodifying], insert the following section. All subsequent sections should be renumbered as appropriate (but they aren’t here for the purposes of the review).]

25.?? Tag specifiers

namespace tag {
struct in { /* implementation-defined */ }; 
struct in1 { /* implementation-defined */ }; 
struct in2 { /* implementation-defined */ }; 
struct out { /* implementation-defined */ }; 
struct out1 { /* implementation-defined */ }; 
struct out2 { /* implementation-defined */ }; 
struct fun { /* implementation-defined */ }; 
struct min { /* implementation-defined */ }; 
struct max { /* implementation-defined */ }; 
struct begin { /* implementation-defined */ }; 
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 (20.15.2) such that TAGGET(\( D \), tag::X, N) names a tagged getter (20.15.2) 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:

\[
\text{struct } _\text{-input-getter } \\
\text{constexpr decltype(auto) in()} \& \{ \text{return get<N>(static_cast<D&>(*this)); } \}; \\
\text{constexpr decltype(auto) in()} \&\& \{ \text{return get<N>(static_cast<D&&>(*this)); } \}; \\
\text{constexpr decltype(auto) in()} \text{ const } \& \{ \text{return get<N>(static_cast<const D&>(*this)); } \}; \\
\]

—end example]
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate pred);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<ProjectedIteratorType_I<Proj>> Pred>
bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<ProjectedIteratorType_I_t<Rng>, Proj>> Pred>
bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});

1 Returns: true if [first,last) is empty or if pred(*i) \textit{INVoke}(pred, \textit{INVoke}(proj, *i)) is true for every iterator i in the range [first,last), and false otherwise.
2 Complexity: At most last - first applications of the predicate and last - first applications of the projection.

25.2.2 Any of [alg.any_of]

template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last, Predicate pred);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<ProjectedIteratorType_I<Proj>> Pred>
bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<ProjectedIteratorType_I_t<Rng>, Proj>> Pred>
bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});

1 Returns: false if [first,last) is empty or if there is no iterator i in the range [first,last) such that pred(*i) \textit{INVoke}(pred, \textit{INVoke}(proj, *i)) is true, and true otherwise.
2 Complexity: At most last - first applications of the predicate and last - first applications of the projection.

25.2.3 None of [alg.none_of]

template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last, Predicate pred);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallablePredicate<ProjectedIteratorType_I<Proj>> Pred>
bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<ProjectedIteratorType_I_t<Rng>, Proj>> Pred>
bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});

1 Returns: true if [first,last) is empty or if pred(*i) \textit{INVoke}(pred, \textit{INVoke}(proj, *i)) is false for every iterator i in the range [first,last), and false otherwise.
2 Complexity: At most last - first applications of the predicate and last - first applications of the projection.

25.2.4 For each [alg.foreach]

template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
template<InputIterator I, Sentinel<I> S, class Proj = identity,  
IndirectCallable<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,  
IndirectCallable<IteratorType iterator_t<Rng>, Proj>> Fun>  
tagged_pair<tag::in(iterator_t<Rng>, Proj), tag::fun(Fun)>  
for_each(Rng& rng, Fun f, Proj proj = Proj{});

1
Requires: Function shall meet the requirements of MoveConstructible (Table 20 19.4.4). [Note:  
Function need not meet the requirements of CopyConstructible (Table 21 19.4.5). —end note]

2
Effects: Applies \( f \) to the result of dereferencing every iterator. Calls \( \text{INVOKE}(f, \text{INVOKE}(proj, *i)) \) for  
every iterator \( i \) in the range \( [\text{first}, \text{last}) \), starting from \( \text{first} \) and proceeding to \( \text{last} - 1 \). [Note:  
If the type of \( \text{first} \) satisfies the requirements of a mutable iterator, result of \( \text{INVOKE}(proj, *i) \) is a  
mutable reference, \( f \) may apply nonconstant functions through the dereferenced iterator. —end note]

3
Returns: \( \text{std::move}(f) \{\text{last}, \text{std::move}(f)\} \).

4
Complexity: Applies \( f \) and \( \text{proj} \) exactly \( \text{last} - \text{first} \times \) times.

5
Remarks: If \( f \) returns a result, the result is ignored.

25.2.5 Find [alg.find]

template<class InputIterator, class T>  
InputIterator find(InputIterator first, InputIterator last,  
const T& value);

template<class InputIterator, class Predicate>  
InputIterator find_if(InputIterator first, InputIterator last,  
Predicate pred);

template<class InputIterator, class Predicate>  
InputIterator find_if_not(InputIterator first, InputIterator last,  
Predicate pred);

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>  
requires IndirectCallableRelation<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 IndirectCallableRelation<equal_to<>, Projected<IteratorType iterator_t<Rng>, Proj>, const T*>()  
IteratorType safe_iterator_t<Rng>  
find(Rng& rng, const T& value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,  
IndirectCallablePredicate<Projected<I, Proj>> Pred>  
I find_if(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,  
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>  
IteratorType safe_iterator_t<Rng>  
find_if(Rng& rng, Pred pred, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,  
IndirectCallablePredicate<Projected<I, Proj>> Pred>  
I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

§ 25.2.5
template<InputRange Rng, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType_iterator_t<Rng>, Proj> >> Pred>
IteratorType_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:
* i == value, pred(*i) != false, pred(*i) == false
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.

25.2.6 Find end [alg.find.end]

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
Sentinel<I2> S2, class Proj = identity,
IndirectCallableRelation<I2, #projected<I1, Proj>> Pred = equal_to<>>
I1
find_end(I1 first1, S1 last1, I2 first2, S2 last2,
Pred pred = Pred{}, Proj proj = Proj{});

25.2.7 Find first [alg.find.first.of]

template<class InputIterator, class ForwardIterator>
InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2);
template<class InputIterator, class ForwardIterator, 
    class BinaryPredicate>
InputIterator
    find_first_of(InputIterator first1, InputIterator last1, 
    ForwardIterator first2, ForwardIterator last2, 
    BinaryPredicate pred);

template<InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2, 
    class Proj1 = identity, class Proj2 = identity, 
    IndirectCallablePredicate<Projected1, Proj1>, Projected2, 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<ForwardRange Rng1, ForwardRange Rng2, class Proj1 = identity, 
//    class Proj2 = identity, 
//    IndirectCallableRelation<Projected1, Proj1>, 
//    IndirectCallableRelation<Projected2, Proj2>> Pred = equal_to<>>
//    I1
//    find_first_of(Rng1 & rng1, Rng2 & rng2, Pred pred = Pred{}, 
//    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

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: i == j, pred(*i, *j) != false INVOKE(pred, 
    INVOKE(proj1, *i), INVOKE(proj2, *j)) != false. Returns last1 if [first2, last2) is empty or if no such iterator is found.

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

25.2.8 Adjacent find [alg.adjacent.find]

template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last, 
    BinaryPredicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectCallableRelation<Projected, Proj>> Pred = equal_to<>>
I
    adjacent_find(I first, S last, Pred pred = Pred{}, 
    Proj proj = Proj{});

//template<ForwardRange Rng, class Proj = identity, 
//    IndirectCallableRelation<Projected, Proj>> Pred = equal_to<>>
//    I
//    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 conditions holds: i == *(i + 1), pred(*i, *(i + 1)) != false INVOKE(pred, 
    INVOKE(proj, *i), INVOKE(proj, *(i + 1))) != false. Returns last if no such iterator is found.

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

### 25.2.9 Count

[alg.count]

```cpp
template<class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
    typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectCallableRelation<
        equal_to<
            projected<I, Proj>, const T*>
    >
    DifferenceType
        difference_type_t<I>
count(I first, S last, const T& value, Proj proj = Proj{});

template<InputRange Rng, class T, class Proj = identity>
    requires IndirectCallableRelation<
        equal_to<
            projected<
                IteratorType iterator_t<Rng>, Proj>, const T*>
    >
    DifferenceType
        difference_type_t<
            IteratorType iterator_t<Rng>>
count(Rng&& rng, const T& value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectCallablePredicate<
        projected<I, Proj>>, Pred>
    DifferenceType
        difference_type_t<I>
count_if(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class T, class Proj = identity,
    IndirectCallablePredicate<
        projected<
            IteratorType iterator_t<Rng>, Proj>>, Pred>
    DifferenceType
        difference_type_t<
            IteratorType iterator_t<Rng>>
count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1. **Effects:** Returns the number of iterators \( i \) in the range \([\text{first, last})\) for which the following corresponding conditions hold: \( *i == \text{value}, \text{INVOKE}(\text{proj}, *i) == \text{value}, \text{INVOKE}(\text{pred}, \text{INVOKE}(\text{proj}, *i)) != \text{false} \).

2. **Complexity:** Exactly \( \text{last - first} \) applications of the corresponding predicate and projection.

### 25.2.10 Mismatch

[mismatch]

```cpp
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
    class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2);
```

§ 25.2.10
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred);

// D.10 (deprecated):
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2, class Proj1 = identity, class Proj2 = identity, IndirectCallablePredicate<Projected<I1, Proj1>, Projected<I2, Proj2>>, Pred = equal_to<>>
tagged_pair<tag::in1(I1), tag::in2(I2)>
mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<InputRange Rng1, WeakInputIterator I2, class Proj1 = identity, class Proj2 = identity, IndirectCallablePredicate<Projected<IteratorType_iterator_t<Rng1>, Proj1>, Projected<IteratorType_iterator_t<Rng2>, Proj2>>, Pred = equal_to<>>
tagged_pair<tag::in1(IteratorType_safe_iterator_t<Rng1>), tag::in2(I2)>
mismatch(Rng1& rng1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, class Proj1 = identity, class Proj2 = identity, IndirectCallablePredicate<Projected<IteratorType_iterator_t<Rng1>, Proj1>, Projected<IteratorType_iterator_t<Rng2>, 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, IndirectCallablePredicate<Projected<IteratorType_iterator_t<Rng1>, Proj1>, Projected<IteratorType_iterator_t<Rng2>, Proj2>>, Pred = equal_to<>>
tagged_pair<tag::in1(IteratorType_safe_iterator_t<Rng1>), tag::in2(IteratorType_safe_iterator_t<Rng2>)>
mismatch(Rng1& rng1, Rng2& rng2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

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:

(2.1) j is in the range [first2, last2).

(2.2) !(i == *(first2 + (i - first1)))

(2.3) pred(*i, *(first2 + (i - first1))) == false

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.

25.2.11 Equal

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,

§ 25.2.11 145
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
BinaryPredicate pred);

// D.10 (deprecated):
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
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, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template<InputRange Rng1, WeakInputIterator I2, class Pred = equal_to<>,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<IteratorType iterator t<Rng1>, I2, Pred, Proj1, Proj2>()
bool equal(Rng1&& rng1, I2 first2, 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<IteratorType iterator t<Rng1>, IteratorType iterator t<Rng2>, Pred, Proj1, Proj2>()
bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if for every iterator i in the range [first1, last1) the following corresponding condition holds: *(i + (i - first1)) != false INVOKE(pred, INVOKE(proj1, *i), INVOKE(proj2, *(first2 + (i - first1)))) != false. Otherwise, returns false.

Complexity: No applications of the corresponding predicate and projections if InputIterator1 and InputIterator2 meet the requirements of random access iterators I1 and S1 model SizedIteratorRange<I1, S1>() is satisfied, and I2 and S2 model SizedIteratorRange<I2, S2>() is satisfied, and last1 - first1 != last2 - first2. Otherwise, at most min(last1 - first1, last2 - first2) applications of
the corresponding predicate and projections.

### 25.2.12 Is permutation

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>`template&lt;class ForwardIterator1, class ForwardIterator2&gt;</td>
</tr>
<tr>
<td>bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,</td>
</tr>
<tr>
<td>ForwardIterator2 first2);</td>
</tr>
<tr>
<td>`template&lt;class ForwardIterator1, class ForwardIterator2,</td>
</tr>
<tr>
<td>class BinaryPredicate&gt;</td>
</tr>
<tr>
<td>bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,</td>
</tr>
<tr>
<td>ForwardIterator2 first2, BinaryPredicate pred);</td>
</tr>
<tr>
<td>`template&lt;class ForwardIterator1, class ForwardIterator2&gt;</td>
</tr>
<tr>
<td>bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,</td>
</tr>
<tr>
<td>ForwardIterator2 first2, ForwardIterator2 last2);</td>
</tr>
<tr>
<td>`template&lt;class ForwardIterator1, class ForwardIterator2,</td>
</tr>
<tr>
<td>class BinaryPredicate&gt;</td>
</tr>
<tr>
<td>bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,</td>
</tr>
<tr>
<td>ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);</td>
</tr>
</tbody>
</table>

// D.10 (deprecated):
<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>`template&lt;ForwardIterator I1, Sentinel&lt;I1&gt; S1, ForwardIterator I2,</td>
</tr>
<tr>
<td>class Pred = equal_to&lt;&gt;, class Proj1 = identity, class Proj2 = identity&gt;</td>
</tr>
<tr>
<td>requires IndirectlyComparable&lt;I1, I2, Pred, Proj1, Proj2&gt;()</td>
</tr>
<tr>
<td>bool is_permutation(I1 first1, S1 last1, I2 first2,</td>
</tr>
<tr>
<td>Pred pred = Pred{},</td>
</tr>
<tr>
<td>Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});</td>
</tr>
</tbody>
</table>

// D.10 (deprecated):
<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>`template&lt;ForwardRange Rng1, ForwardIterator I2, class Pred = equal_to&lt;&gt;,</td>
</tr>
<tr>
<td>class Proj1 = identity, class Proj2 = identity&gt;</td>
</tr>
<tr>
<td>requires IndirectlyComparable&lt;IteratorType iterator_t&lt;Rng1&gt;, I2, Pred,</td>
</tr>
<tr>
<td>Proj1, Proj2&gt;()</td>
</tr>
<tr>
<td>bool is_permutation(Rng1&amp; rng1, I2 first2, Pred pred = Pred{},</td>
</tr>
<tr>
<td>Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>`template&lt;ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to&lt;&gt;,</td>
</tr>
<tr>
<td>class Proj1 = identity, class Proj2 = identity&gt;</td>
</tr>
<tr>
<td>requires IndirectlyComparable&lt;IteratorType iterator_t&lt;Rng1&gt;,</td>
</tr>
<tr>
<td>IteratorType iterator_t&lt;Rng2&gt;, Pred, Proj1, Proj2&gt;()</td>
</tr>
<tr>
<td>bool is_permutation(Rng1&amp; rng1, Rng2&amp; rng2, Pred pred = Pred{},</td>
</tr>
<tr>
<td>Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});</td>
</tr>
</tbody>
</table>

1. **Requires:** `ForwardIterator1` and `ForwardIterator2` shall have the same value type. The comparison function shall be an equivalence relation.
2. **Remarks:** If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.
3. **Returns:** If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2, first2 + (last1 - first1)]`, beginning...
with \texttt{ForwardIterator1} begin, such that \texttt{equal(first1, last1, begin, pred, proj1, proj2)} returns true or \texttt{equal(first1, last1, begin, pred)} returns true; otherwise, returns false.

**Complexity:** No applications of the corresponding predicate and projections if \texttt{ForwardIterator1} and \texttt{ForwardIterator2} meet the requirements of random access iterators \texttt{I1} and \texttt{S1} model \texttt{SizedIteratorRange} and \texttt{I2} and \texttt{S2} model \texttt{SizedIteratorRange} is satisfied, and last1 - first1 != last2 - first2. Otherwise, exactly distance(first1, last1) applications of the corresponding predicate and projections if equal(first1, last1, first2, last2, pred, proj1, proj2) would return true if \texttt{proj1} was not given in the argument list or equal(first1, last1, first2, last2, pred) would return true if \texttt{proj1} was given in the argument list; otherwise, at worst $\Theta(N^2)$, where $N$ has the value distance(first1, last1).

### 25.2.13 Search

#### template<class ForwardIterator1, class ForwardIterator2>

\begin{verbatim}
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);
\end{verbatim}

#### template<class ForwardIterator1, class ForwardIterator2,
            class BinaryPredicate>

\begin{verbatim}
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPredicate pred);
\end{verbatim}

#### template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
            Sentinel<I2> S2, class Pred = equal_to<>,
            class Proj1 = identity, class Proj2 = identity>

\begin{verbatim}
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{});
\end{verbatim}

#### template<ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
            class Proj1 = identity, class Proj2 = identity>

\begin{verbatim}
requires IndirectlyComparable<
iterator_type_t<Rng1>, iterator_type_t<Rng2>, Pred, Proj1, Proj2>()
iterator_type safe_iterator_t<Rng1>
search(Rng1& rng1, Rng2& rng2, Pred pred = Pred{},
       Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
\end{verbatim}

**Effects:** Finds a subsequence of equal values in a sequence.

**Returns:** The first iterator in the range [first1, last1 - (last2 - first2)) such that for every non-negative integer $n$ less than last2 - first2 the following corresponding conditions hold: $(i + n) == *(first2 + n)$, pred(*((i + n) + first2 + n)) != false \texttt{INVOKE} (pred, \texttt{INVOKE} (proj1, *((i + n) + first2 + n)) != false. Returns first1 if [first2, last2) is empty, otherwise returns last1 if no such iterator is found.

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

#### template<class ForwardIterator, class Size, class T>

\begin{verbatim}
ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count,
                         const T& value);
\end{verbatim}

§ 25.2.13
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator
search_n(ForwardIterator first, ForwardIterator last, Size count,
const T& value, BinaryPredicate pred);

template<ForwardIterator I, Sentinel<I> S, class T, class Pred = equal_to<>, class Proj = identity>
requires IndirectlyComparable<I*, const T*, Pred, Proj>()
I
search_n(I first, S last, DifferenceType difference_type_t<I> count, 
cost T& value, Pred pred = Pred{}, 
Proj proj = Proj{});

template<ForwardRange Rng, class T, class Pred = equal_to<>, 
class Proj = identity>
requires IndirectlyComparable<IteratorType iterator_t<Rng>, const T*, Pred, Proj>()
IteratorType
search_n(Rng& rng, DifferenceType difference_type_t<IteratorType iterator_t<Rng>> count,
const T& value, Pred pred = Pred{}, Proj proj = Proj{});

Requires: The type Size shall be convertible to integral type (4.7, 12.3).
Effects: Finds a subsequence of equal values in a sequence.
Returns: The first iterator i in the range [first,last-count) such that for every non-negative integer n less than count the following corresponding conditions hold: *(i + n) == value, pred(*((i + n).value)) != false INVOKE(pred, INVOKE(proj, *(i + n)), value) != false. Returns last if no such iterator is found.
Complexity: At most last - first applications of the corresponding predicate and projection.

§ 25.3 Mutating sequence operations [alg.modifying.operations] [alg.copy]

25.3.1 Copy

template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, 
OutputIterator result);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
requires IndirectlyCopyable<I, O>()
tagged_pair::in(I), tag::out(O)>
copy(I first, S last, 0 result);

template<InputRange Rng, WeaklyIncrementable O>
requires IndirectlyCopyable<IteratorType iterator_t<Rng>, O>()
tagged_pair::in(iteratorType safe_iterator_t<Rng>), tag::out(O)>
copy(Rng& rng, 0 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: result + (last - first)(last, result + (last - first)).
Requires: result shall not be in the range [first,last).
Complexity: Exactly last - first assignments.
template<class InputIterator, class Size, class OutputIterator>
    OutputIterator copy_n(InputIterator first, Size n,
                           OutputIterator result);

    template<WeakInputIterator I, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>()
    tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, DifferenceType difference_type_t<I> n, 0 result);

5   Effects: For each non-negative integer i < n, performs *(result + i) = *(first + i).
6   Returns: result + n{first + n, result + n}.
7   Complexity: Exactly n assignments.

template<class InputIterator, class OutputIterator, class Predicate>
    OutputIterator copy_if(InputIterator first, InputIterator last,
                            OutputIterator result, Predicate pred);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
          IndirectCallablePredicate<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{});

7   Requires: The ranges [first,last) and [result,result + (last - first)) shall not overlap.
8   Effects: Copies all of the elements referred to by the iterator i in the range [first,last) for which
9   pred(*i)\INVOKE(pred, \INVOKE(proj, *i)) is true.
10  Returns: The end of the resulting range {last, result + (last - first)}.
11  Complexity: Exactly last - first applications of the corresponding predicate and projection.
12  Remarks: Stable (17.6.5.7).

template<class BidirectionalIterator1, class BidirectionalIterator2>
    BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyCopyable<I1, I2>()
    tagged_pair<tag::in(I1), tag::inout(I2)>
    copy_backward(I1 first, I1 last, I2 result);

template<BidirectionalRange Rng, BidirectionalIterator I>
    requires IndirectlyCopyable<IteratorType iterator_t<Rng>, I>()
    tagged_pair<tag::in(I), Tag<IteratorType safe_iterator_t<Rng>>, tag::inout(I)>
    copy_backward(Rng& rng, I result);
Effects: Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result} - (\text{last} - \text{first}), \text{result})\) starting from \(\text{last} - 1\) and proceeding to \(\text{first}\). For each positive integer \(n \leq (\text{last} - \text{first})\), performs \(*(*\text{result} - n) = *(\text{last} - n)\).

Requires: \text{result} shall not be in the range \([\text{first}, \text{last}]\).

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

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

25.3.2 Move

\[\text{alg.move}\]

\[
\begin{align*}
\text{template}&\langle\text{class InputIterator}, \text{class OutputIterator}\rangle \\
&\quad \text{OutputIterator move(InputIterator first, InputIterator last,} \\
&\quad \quad \text{OutputIterator result);} \\
\text{template}&\langle\text{InputIterator I, Sentinel\langle I\rangle S, WeaklyIncrementable O}\rangle \\
&\quad \text{requires IndirectlyMovable\langle I, O\rangle}() \\
&\quad \text{tagged\_pair\langle tag::in(I), tag::out(O)\rangle} \\
&\quad \text{move(I first, S last, O result);} \\
\text{template}&\langle\text{InputRange Rng, WeaklyIncrementable O}\rangle \\
&\quad \text{requires IndirectlyMovable\langle IteratorType \_iterator\_t<Rng>, O\rangle}() \\
&\quad \text{tagged\_pair\langle tag::in(IteratorType \_safe\_iterator\_t<Rng>), tag::out(O)\rangle} \\
&\quad \text{move(Rng\& rng, O result);} \\
\end{align*}
\]

Effects: Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result, result} + (\text{last} - \text{first}))\) starting from \(\text{first}\) and proceeding to \(\text{last}\). For each non-negative integer \(n < (\text{last} - \text{first})\), performs \(*(*\text{result} + n) = \text{std::move}(*(*\text{first} + n))\).

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

Requires: \text{result} shall not be in the range \([\text{first, last})\).

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

\[\text{template}\langle\text{class BidirectionalIterator1, class BidirectionalIterator2}\rangle \\
&\quad \text{BidirectionalIterator2} \\
&\quad \text{move\_backward(BidirectionalIterator1 first,} \\
&\quad \quad \text{BidirectionalIterator1 last,} \\
&\quad \quad \text{BidirectionalIterator2 result);} \\
\text{template}&\langle\text{BidirectionalIterator I1, Sentinel\langle I1\rangle S1, BidirectionalIterator I2}\rangle \\
&\quad \text{requires IndirectlyMovable\langle I1, I2\rangle}() \\
&\quad \text{tagged\_pair\langle tag::in\_1(I1), tag::in\_2out(I2)\rangle} \\
&\quad \text{move\_backward(I1 first, S1 last, I2 result);} \\
\text{template}&\langle\text{BidirectionalRange Rng, BidirectionalIterator I}\rangle \\
&\quad \text{requires IndirectlyMovable\langle IteratorType \_iterator\_t<Rng>, I\rangle}() \\
&\quad \text{tagged\_pair\langle tag::in\_1(IteratorType \_safe\_iterator\_t<Rng>), tag::in\_1out(I)\rangle} \\
&\quad \text{move\_backward(Rng\& rng, I result);} \\
\end{align*}
\]

Effects: Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result} - (\text{last} - \text{first}), \text{result})\) starting from \(\text{last} - 1\) and proceeding to \(\text{first}\). For each positive integer \(n \leq (\text{last} - \text{first})\), performs \(*(*\text{result} - n) = \text{std::move}(*(*\text{last} - n))\).

Requires: \text{result} shall not be in the range \([\text{first, last})\).

4) \text{copy\_backward} should be used instead of \text{copy} when \text{last} is in the range \([\text{result} - (\text{last} - \text{first}), \text{result})\).

5) \text{move\_backward} should be used instead of \text{move} when \text{last} is in the range \([\text{result} - (\text{last} - \text{first}), \text{result})\).
Returns: \( \text{result} - (\text{last} - \text{first}) \{ \text{last}, \text{result} - (\text{last} - \text{first}) \}. \)

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

### 25.3.3 swap

```cpp
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

// D.10 (deprecated):
```cpp
template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in1(I1), tag::in2(I2)>
    swap_ranges(I1 first1, S1 last1, I2 first2);
```

// D.10 (deprecated):
```cpp
template<ForwardRange Rng, ForwardIterator I>
requires IndirectlySwappable<IteratorType iterator_t<Rng>, I>()
tagged_pair<tag::in1(IteratorType safe_iterator_t<Rng>), tag::in2(I)>
    swap_ranges(Rng& rng1, I first2);
```

```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<IteratorType iterator_t<Rng1>, iteratorType iterator_t<Rng2>>()
tagged_pair<tag::in1(iteratorType safe_iterator_t<Rng1>), tag::in2(iteratorType safe_iterator_t<Rng2>)>
    swap_ranges(Rng1& rng1, Rng2& rng2);
```

Effects: For the first two overloads, let \( \text{last2} \) be \( \text{first2} + (\text{last1} - \text{first1}) \). For each non-negative integer \( n < (\text{last1} - \text{first1}) \), \( n < \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \) performs: \( \text{swap}(\ast (\text{first1} + n), \ast (\text{first2} + n)) \).

Requires: The two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{first2} + (\text{last1} - \text{first1})) \) shall not overlap. \( \ast (\text{first1} + n) \) shall be swappable with \( (19.2.10) \ast (\text{first2} + n) \).

Returns: \( \text{first2} + (\text{last1} - \text{first1}) \{ \text{first1} + n, \text{first2} + n \} \), where \( n = \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \).

Complexity: Exactly \( \text{last1} - \text{first1} \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \) swaps.

```cpp
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Effects: \( \text{swap}(\ast a, \ast b) \).

Requires: \( a \) and \( b \) shall be dereferenceable. \( \ast a \) shall be swappable with \( (19.2.10) \ast b \).

### 25.3.4 Transform

```cpp
template<class InputIterator, class OutputIterator,
        class UnaryOperation>
OutputIterator
transform(InputIterator first, InputIterator last,
          OutputIterator result, UnaryOperation op);
```

§ 25.3.4
```c++
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, OutputIterator result,
         BinaryOperation binary_op);

template<InputIterator I, Sentinel<I> S,
         WeaklyIncrementable O,
         class F, class Proj = identity,
         typename Proj::Projected<I, Proj>
         O>
requires Writable<O, typename Proj::Projected<I, Proj>::indirect_result_of_t<F&>
tagged_pair<tag::in(I), tag::out(O)>
transform(I first, S last, O result, F op, Proj proj = Proj{});

// D.10 (deprecated):  

// D.10 (deprecated):
```
template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class F, class Proj1 = identity, class Proj2 = identity>
IndirectCallableRelation
Projected<IteratorType<Rng1>, Proj1>,
Projected<IteratorType<Rng2>, Proj2>> F,
WeakOutputIterator<IndirectCallableResultType<F>,
Projected<IteratorType<Rng1>, Proj1>,
Projected<IteratorType<Rng2>, Proj2>>> O
requires Writable<O, IndirectCallableResultType<F, Projected<IteratorType<Rng1>, Proj1>, Projected<IteratorType<Rng2>, Proj2>>>}

replace(I first1, S last1, I2 first2, S2 last2, 0 result,
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Effects: Assigns through every iterator i in the range \([\text{result}, \text{result} + (\text{last1} - \text{first1}) N)\) a new corresponding value equal to \(\text{op}((\text{first1} + (i - \text{result})))\) or \(\text{binary_op}((\text{first1} + (i - \text{result})), (\text{first2} + (i - \text{result})))\).

Requires: \(\text{op}\) and \(\text{binary_op}\) shall not invalidate iterators or subranges, or modify elements in the ranges \([\text{first1}, \text{last} \text{first1} + N], [\text{first2}, \text{first2} + (\text{last1} - \text{first1}) N]\), and \([\text{result}, \text{result} + (\text{last1} - \text{first1}) N]\).

Returns: \(\text{result} + (\text{last1} - \text{first1}) (\text{first1} + N, \text{result} + N)\) or \(\text{make_tagged_tuple}(<\text{in1}, \text{tag::in2}, \text{tag::out}>(\text{first1} + N, \text{first2} + N, \text{result} + N))\).

Complexity: Exactly \(\text{last1} - \text{first1} N\) applications of \(\text{op}\) or \(\text{binary_op}\).

Remarks: \(\text{result}\) may be equal to \(\text{first1}\) in case of unary transform, or to \(\text{first1}\) or \(\text{first2}\) in case of binary transform.

25.3.5 Replace

\[\text{alg.replace}\]

\text{template}<\text{class ForwardIterator, class T}>
\text{void replace}(\text{ForwardIterator first}, \text{ForwardIterator last},
\text{const T& old_value, const T& new_value});

\text{template}<\text{class ForwardIterator, class Predicate, class T}>
\text{void replace_if}(\text{ForwardIterator first}, \text{ForwardIterator last},
\text{Predicate pred, const T& new_value});

\text{template}<\text{ForwardIterator I, Sentinel<I> S, class T1, Semiregular class T2, class Proj = identity}>
\text{requires Writable<I, T2> & &
IndirectCallableRelation<T1, Proj>, const T1>}
\text{I}
\text{replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{})};

---

6) The use of fully closed ranges is intentional.
template<ForwardRange Rng, class T1, Semiregular class T2, class Proj = identity>
requires Writable<IteratorType safe_iterator_t<Rng>, T1> &&
IndirectCallableRelation<Equal_to<>, Projected<IteratorType safe_iterator_t<Rng>, Proj>>, const T1*>(
iterator safe_iterator_t<Rng>
replace(Rng& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, Semiregular class T, class Proj = identity,
IndirectCallablePredicate<Projected<I, Proj>>, Pred>
requires Writable<I, T>() &&
IndirectCallableRelation<Equal_to<>, Projected<I, Proj>>, const T*>(
I
replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template<ForwardRange Rng, Semiregular class T, class Proj = identity,
IndirectCallablePredicate<Projected<IteratorType safe_iterator_t<Rng>, Proj>>, Pred>
requires Writable<IteratorType safe_iterator_t<Rng>, T>() &&
IndirectCallableRelation<Equal_to<>, Projected<IteratorType safe_iterator_t<Rng>, Proj>>, const T*>(
IteratorType safe_iterator_t<Rng>
replace_if(Rng& rng, Pred pred, const T& new_value, Proj proj = Proj{});

1 Requires: The expression *first = new_value shall be valid.

2 Effects: Substitutes elements referred by the iterator i in the range [first,last) with new_value,
when the following corresponding conditions hold: *i == old_value INVOKE(proj, *i) == old_value,
pred(*i) != false INVOKE(pred, INVOKE(proj, *i)) != false.

3 Returns: last.

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

template<class InputIterator, class OutputIterator, class T>
OutputIterator
replace_copy(InputIterator first, InputIterator last,
OutputIterator result,
const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator
replace_copy_if(InputIterator first, InputIterator last,
OutputIterator result,
Predicate pred, const T& new_value);

template<InputIterator I, Sentinel<I> S, class T1, Semiregular class T2, WeakOutputIterator<T2> O,
class Proj = identity>
requires IndirectlyCopyable<I, O>() &&
IndirectCallableRelation<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, Semiregular class T2, WeakOutputIterator<T2> O,
class Proj = identity>
requires IndirectlyCopyable<IteratorType safe_iterator_t<Rng>, O>() &&
IndirectCallableRelation<Equal_to<>, Projected<IteratorType safe_iterator_t<Rng>, Proj>>, const T1*>(
tagged_pair<tag::in(IteratorType 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, Semiregular class T, WeakOutputIterator<T> O,
class Proj = identity, IndirectCallablePredicate<Projected<I, Proj>>, Pred>
§ 25.3.5 155
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, Semiregular class T, WeakOutputIterator<T> O, class Proj = identity, IndirectCallablePredicate<Projected<IteratorTypeIterator_t<Rng>, Proj>> Pred>
requires IndirectlyCopyable<IteratorTypeIterator_t<Rng>, O>()
tagged_pair<tag::in(IteratorTypeIterator_safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng& rng, O result, Pred pred, const T& new_value,
    Proj proj = Proj());

5  Requires: The results of the expressions *first and new_value shall be writable to the result output iterator. The ranges [first,last) and [result,result + (last - first)) shall not overlap.

6  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:
   *(first + (i - result)) == old_value
   pred(*(first + (i - result))) != false
   INVOKE(proj, *(first + (i - result))) == old_value
   INVOKE(pred, INVOKE(proj, *(first + (i - result)))) != false

7  Returns: {last, result + (last - first)}.
8  Complexity: Exactly last - first applications of the corresponding predicate and projection.

25.3.6 Fill [alg.fill]

template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);

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

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

template<Semiregular class T, WeakOutputIterator<T> O>
    O fill_n(O first, DifferenceTypeDifference_type_t<O> n, const T& value);

1  Requires: The expression value shall be writable to the output iterator. The type Size shall be convertible to an integral type (4.7, 12.3).
2  Effects: The first algorithm fill assigns value through all the iterators in the range [first,last). The second algorithm fill_n assigns value through all the iterators in the range [first,first + n) if n is positive, otherwise it does nothing.
3  Returns: fill returns last. fill_n returns first + n for non-negative values of n and first for negative values.
4  Complexity: Exactly last - first, n, or 0 assignments, respectively.

§ 25.3.6
25.3.7 Generate

类型的 Forwards Iterator, class Generator>
    void generate(ForwardIterator first, ForwardIterator last, 
    Generator gen);

template<class OutputIterator, class Size, class Generator>
    OutputIterator generate_n(OutputIterator first, Size n, Generator gen);

template<Function F, OutputIterator<ResultType<F>, result_of_t<F&&>> O, 
    Sentinel<O> S>
    O generate(O first, S last, F gen);

template<Function F, OutputRange<ResultType<F>, result_of_t<F&&>> Rng>
    IteratorType safe_iterator_t<Rng>
    generate(Rng& rng, F gen);

template<Function F, WeakOutputIterator<ResultType<F>, result_of_t<F&&>> O>
    O generate_n(O first, DifferenceType difference_type_t<O> n, F gen);

1 Effects: The first algorithm generate invokes the function object gen and assigns the return value of gen through all the iterators in the range [first, last). The second algorithm generate_n invokes the function object gen and assigns the return value of gen through all the iterators in the range [first, first + n) if n is positive, otherwise it does nothing.

2 Requires: gen takes no arguments, Size shall be convertible to an integral type (4.7, 12.3).

3 Returns: generate returns last. generate_n returns first + n for non-negative values of n and first for negative values.

4 Complexity: Exactly last - first, n, or 0 invocations of gen and assignments, respectively.

25.3.8 Remove

template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires Permutable<I>()
    IndirectCallableRelation(equal_to<>, ProjectedI, 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<IteratorType iterator_t<Rng>>() &
    IndirectCallableRelation(equal_to<>, Projected<IteratorType iterator_t<Rng>, Proj>, const T*)&>
    IteratorType safe_iterator_t<Rng>
    remove(Rng& rng, const T& value, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectCallablePredicate<ProjectedI, Proj>> Pred>
    requires Permutable<I>()
    I remove_if(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity, 
    IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>>, Pred>
    requires Permutable<IteratorType iterator_t<Rng>>()
    IteratorType safe_iterator_t<Rng>
    remove_if(Rng& rng, Pred pred, Proj proj = Proj{});

1 Requires: The type of *first shall satisfy the MoveAssignable requirements (Table 22).
Effects: Eliminates all the elements referred to by iterator \( i \) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions hold:

\[
* i == \text{value} \Rightarrow \text{INVOK} (\text{proj}, * i) == \text{value}, \text{pred}(* i) != \text{false} \Rightarrow \text{INVOK} (\text{pred}, \text{INVOK} (\text{proj}, * i)) != \text{false}.
\]

Returns: The end of the resulting range.

Remarks: Stable (17.6.5.7).

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

Note: each element in the range \([\text{ret}, \text{last})\), where \text{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.

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator
remove_copy(InputIterator first, InputIterator last, OutputIterator result, const T& value);
```

```cpp
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator
remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);
```

```cpp
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T, class Proj = identity>
requires IndirectlyCopyable<I, O>() &&
IndirectCallableRelation<equal_to<>, Projected<I, Proj>, const T*>(I)
tagged_pair<tag::in(I), tag::out(O)>
remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});
```

```cpp
template<InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
requires IndirectlyCopyable<IteratorType iterator_t<Rng>, O>() &&
IndirectCallableRelation<equal_to<>, Projected<IteratorType iterator_t<Rng>, Proj>, const T*>(iterator_t<Rng>)
tagged_pair<tag::in(iterator_t<Rng>), tag::out(O)>
remove_copy(Rng& rng, O result, const T& value, Proj proj = Proj{});
```

```cpp
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T, class Proj = identity, IndirectCallablePredicate<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{});
```

```cpp
template<InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity, IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
requires IndirectlyCopyable<IteratorType iterator_t<Rng>, O>()
tagged_pair<tag::in(iterator_t<Rng>), tag::out(O)>
remove_copy_if(Rng& rng, O result, Pred pred, Proj proj = Proj{});
```

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

Effects: Copies all the elements referred to by the iterator \( i \) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions do not hold:

\[
* i == \text{value} \Rightarrow \text{INVOK} (\text{proj}, * i) == \text{value}, \text{pred}(* i) != \text{false} \Rightarrow \text{INVOK} (\text{pred}, \text{INVOK} (\text{proj}, * i)) != \text{false}.
\]

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

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

Remarks: Stable (17.6.5.7).
25.3.9 Unique

```cpp
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
        BinaryPredicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
        IndirectCallableRelation<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,
        IndirectCallableRelation<Projected<IteratorType<iterator_t<Rng>>, Proj>> R = equal_to<>>
        requires Permutable<IteratorType<iterator_t<Rng>>>()
        IteratorType<safe_iterator_t<Rng>> unique(Rng& rng, R comp = R{}, Proj proj = Proj{});
```

1. **Effects:** For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator i in the range [first + 1, last) for which the following conditions hold: *(i - 1) == i* \text{INVOKED(proj, *i - 1) == INVOKED(proj, *i)} or \text{pred(*_(i - 1), *i) != false*} \text{INVOKED(proj, *i - 1), INVOKED(proj, *i) != false*}.

2. **Requires:** The comparison function shall be an equivalence relation. The type of *first shall satisfy the MoveAssignable requirements (Table 22).

3. **Returns:** The end of the resulting range.

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

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
        OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
        OutputIterator result, BinaryPredicate pred);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
        class Proj = identity, IndirectCallableRelation<Projected<I, Proj>> R = equal_to<>>
        requires IndirectlyCopyable<I, O>() \&\& (ForwardIterator<I>() \| |
        ForwardIterator<O>() \| |
        Copyable<ValueType<value_type_t<I>>>())
        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,
        IndirectCallableRelation<Projected<IteratorType<iterator_t<Rng>>, Proj>> R = equal_to<>>
        requires IndirectlyCopyable<IteratorType<iterator_t<Rng>>, O>() \&\& (ForwardIterator<IteratorType<iterator_t<Rng>>>() \| |
        Copyable<ValueType<value_type_t<IteratorType<iterator_t<Rng>>>()) \| |
        ForwardIterator<O>() \| |
        Copyable<ValueType<value_type_t<IteratorType<iterator_t<Rng>>>())
        tagged_pair<tag::in(IteratorType<safe_iterator_t<Rng>>), tag::out(O)>
        unique_copy(Rng& rng, O result, R comp = R{}, Proj proj = Proj{});
```

§ 25.3.9
5 \textbf{Requires:} The comparison function shall be an equivalence relation. The ranges \([\text{first, last})\) and \([\text{result, result+(last-first)})\) shall not overlap. The expression \(\text{result} = \text{first}\) shall be valid. If neither InputIterator nor OutputIterator meets the requirements of forward iterator then the value type of InputIterator shall be CopyConstructible (Table 21) and CopyAssignable (Table 22). Otherwise CopyConstructible is not required.

6 \textbf{Effects:} Copies only the first element from every consecutive group of equal elements referred to by the iterator \(i\) in the range \([\text{first, last})\) for which the following corresponding conditions hold: \(\forall i \leq (\text{last - i - 1})\): \(\text{INVOKED}(\text{proj}, \ast i) == \text{INVOKED}(\text{proj}, \ast (i - 1))\) or \(\text{pred}(\ast i, \ast (i - 1)) \neq \text{false}\).

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

8 \textbf{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.

25.3.10 Reverse

\[\text{Reverse} \quad \text{[alg.reverse]}\]

\textbf{template<}} \text{class BidirectionalIterator>}
\text{void reverse(BidirectionalIterator first, BidirectionalIterator last);}\]

\textbf{template<}} \text{BidirectionalIterator I, Sentinel\text{\langle}I\text{\rangle}\ S>}
\text{requires Permutable\text{\langle}I\text{\rangle}\ ()\ I}
\text{I.reverse(I, first, S last);}\]

\textbf{template<}} \text{BidirectionalRange Rng>}
\text{requires Permutable\text{\langle}I\text{\rangle}t\text{\langle}Rng\rangle\ ()\ Rng}
\text{reverse(Rng\&\ rng);}\]

1 \textbf{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 \textbf{Requires:} \(\ast \text{first}\) shall be swappable (19.2.10).

3 \textbf{Returns:} \text{last}.

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

\textbf{template<}} \text{class BidirectionalIterator, class OutputIterator>}
\text{OutputIterator}
\text{reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);}\]

\textbf{template<}} \text{BidirectionalIterator I, Sentinel\text{\langle}I\text{\rangle}\ S, WeaklyIncrementable O>}
\text{requires IndirectlyCopyable\text{\langle}I, O\text{\rangle}\ ()\ tagged\_pair<\text{tag::in(I)}, \text{tag::out(O)}>}
\text{reverse_copy(I, first, S last, O result);}\]

\textbf{template<}} \text{BidirectionalRange Rng, WeaklyIncrementable O>}
\text{requires IndirectlyCopyable\text{\langle}I\text{\rangle}t\text{\langle}Rng\rangle\ O\ ()\ tagged\_pair<\text{tag::in(I)}, \text{tag::out(O)}>}
\text{reverse_copy(Rng\&\ rng, O result);}\]

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

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

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

8 \textbf{Complexity:} Exactly \(\text{last} - \text{first}\) assignments.
25.3.11 Rotate

```
template<class ForwardIterator>
    ForwardIterator rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);

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<IteratorType iterator_t<Rng>>()
    tagged_pair<tag::begin(IteratorType safe_iterator_t<Rng>), tag::end(IteratorType safe_iterator_t<Rng>)>
    rotate(Rng& rng, IteratorType iterator_t<Rng> middle);
```

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

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

3 Remarks: This is a left rotate.

4 Requires: \([first, middle)\) and \([middle, last)\) shall be valid ranges. \( \text{ForwardIterator} \) shall satisfy the requirements of \( \text{ValueSwappable} \) (19.2.10). The type of \( *first \) shall satisfy the requirements of \( \text{MoveConstructible} \) (Table 20) and the requirements of \( \text{MoveAssignable} \) (Table 22).

5 Complexity: At most \( last - first \) swaps.

```
template<class ForwardIterator, class OutputIterator>
    OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);

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<IteratorType iterator_t<Rng>, O>()
    tagged_pair<tag::in(IteratorType safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng& rng, IteratorType iterator_t<Rng> middle, O result);
```

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

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

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

9 Complexity: Exactly \( last - first \) assignments.

25.3.12 Shuffle

```
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
    void shuffle(RandomAccessIterator first, RandomAccessIterator last, UniformRandomNumberGenerator&& g);
```
template<RandomAccessIterator I, Sentinel<I> S, class Gen>
   requires Permutable<I>() && ConvertibleTo<ResultType<Gen> result_of_t<Gen&>()>, DifferenceType difference_type_t<I>()
   UniformRandomNumberGenerator<remove_reference_t<Gen>>()   
   I shuffle(I first, S last, Gen&& g);

template<RandomAccessRange Rng, class Gen>
   requires Permutable<I>() && ConvertibleTo<ResultType<Gen> result_of_t<Gen&>()>, DifferenceType difference_type_t<I>()
   UniformRandomNumberGenerator<remove_reference_t<Gen>>()
   IteratorType safe_iterator_t<Rng>
   shuffle(Rng&& rng, Gen&& g);

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

2 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (19.2.10). The type UniformRandomNumberGenerator shall meet the requirements of a uniform random number generator (26.5.1.3) type whose return type is convertible to iterator_traits<RandomAccessIterator>::difference_type.

3 Complexity: Exactly (last - first) - 1 swaps.

4 Returns: last

5 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.

25.3.13 Partitions

template<class InputIterator, class Predicate>
   bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);

template<InputIterator I, Sentinel<I> S, class Proj = identity,
   IndirectCallablePredicate<Projected<I, Proj>> Pred>
   bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

template<InputRange Rng, class Proj = identity,
   IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>
   bool is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});

1 Requires: InputIterator’s value type shall be convertible to Predicate’s argument type.

2 Returns: true if [first, last) is empty or if [first, last) is partitioned by pred and proj, i.e. if all elements that satisfy pred appear before those that do not, for every i in [first, last).

3 Complexity: Linear. At most last - first applications of pred and proj.

template<class ForwardIterator, class Predicate>
   ForwardIterator partition(ForwardIterator first,
   ForwardIterator last, Predicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
   IndirectCallablePredicate<Projected<I, Proj>> Pred>
   requires Permutable<I>()
   I partition(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
   IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>

§ 25.3.13
requires Permutable<IteratorType iterator_t<Rng>>()

IteratorType safe_iterator_t<Rng>

partition(Rng & rng, Pred pred, Proj proj = Proj());

Effects: Places all the elements in the range \([\text{first}, \text{last})\) that satisfy \text{pred} before all the elements that do not satisfy it.

Effects: Permutates the elements in the range \([\text{first}, \text{last})\) such that there exists and iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{INVOKE}(\text{pred}, \text{INVOKE}(\text{proj}, *j)) \neq \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\) \(\text{INVOKE}(\text{pred}, \text{INVOKE}(\text{proj}, *k)) \neq \text{false}\).

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

Requires: ForwardIterator shall satisfy the requirements of ValueSwappable (19.2.10).

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

template<class BidirectionalIterator, class Predicate>

BidirectionalIterator

stable_partition(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);

template<BidirectionalIterator I, Sentinel< I > S, class Proj = identity,
IndirectCallablePredicate<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,
IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>

requires Permutable<IteratorType iterator_t<Rng>>()

IteratorType safe_iterator_t<Rng>

stable_partition(Rng & rng, Pred pred, Proj proj = Proj());

Effects: Places all the elements in the range \([\text{first}, \text{last})\) that satisfy \text{pred} before all the elements that do not satisfy it.

Effects: Permutates the elements in the range \([\text{first}, \text{last})\) such that there exists and iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{INVOKE}(\text{pred}, \text{INVOKE}(\text{proj}, *j)) \neq \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\) \(\text{INVOKE}(\text{pred}, \text{INVOKE}(\text{proj}, *k)) \neq \text{false}\).

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

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (19.2.10). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

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

template<class InputIterator, class OutputIterator1,
class OutputIterator2, class Predicate>

pair<OutputIterator1, OutputIterator2>

partition_copy(InputIterator first, InputIterator last,
OutputIterator1 out_true, OutputIterator2 out_false,

§ 25.3.13 163
Predicate pred);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,  
class Proj = identity, IndirectCallablePredicate<Projected<I, Proj>> Pred>  
requires IndirectlyCopyable<I, 01>() && IndirectlyCopyable<I, 02>()  
tagged_tuple::in(I), tag::out1(01), tag::out2(02)>  
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, IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>  
requires IndirectlyCopyable<IteratorType iterator_t<Rng>, 01>() &&  
IndirectlyCopyable<IteratorType iterator_t<Rng>, 02>()  
tagged_tuple::in(IteratorType safe_iterator_t<Rng>), tag::out1(01), tag::out2(02)>  
partition_copy(Rng& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});

12  Requires: InputIterator’s value type shall be CopyAssignable, and shall be writable to the out_true and out_false Output iterators, and shall be convertible to Predicate’s argument type. The input range shall not overlap with either of the output ranges.

13  Effects: For each iterator i in [first,last), copies *i to the output range beginning with out_true if pred(*i)) INVOC(p, INVOC(p, *i)) is true, or to the output range beginning with out_false otherwise.

14  Returns: A tuple p such that get<0>(p) is last, p.first get<1>(p) is the end of the output range beginning at out_true and p.second get<2>(p) is the end of the output range beginning at out_false.

15  Complexity: Exactly last - first applications of pred and proj.

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,  
class Proj = identity, IndirectCallablePredicate<Projected<I, Proj>> Pred>  
requires IndirectlyMovable<I, 01>() && IndirectlyMovable<I, 02>()  
tagged_tuple::in(I), tag::out1(01), tag::out2(02)>  
partition_move(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, IndirectCallablePredicate<Projected<IteratorType iterator_t<Rng>, Proj>> Pred>  
requires IndirectlyMovable<IteratorType iterator_t<Rng>, 01>() &&  
IndirectlyMovable<IteratorType iterator_t<Rng>, 02>()  
tagged_tuple::in(IteratorType safe_iterator_t<Rng>), tag::out1(01), tag::out2(02)>  
partition_move(Rng& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});

16  Requires: The input range shall not overlap with either of the output ranges.

17  Effects: For each iterator i in [first,last), moves *i to the output range beginning with out_true if INVOC(p, INVOC(p, *i)) is true, or to the output range beginning with out_false otherwise.

18  Returns: A tuple p such that get<0>(p) is last, get<1>(p) is the end of the output range beginning at out_true and get<2>(p) is the end of the output range beginning at out_false.

19  Complexity: Exactly last - first applications of pred and proj.
template<class ForwardIterator, class Predicate>
    ForwardIterator partition_point(ForwardIterator first, 
    ForwardIterator last, 
    Predicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
    I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity, 
    IndirectCallablePredicate<Projected<IteratorType_iterator_t<Rng>, Proj>> Pred>
    IteratorType safe_iterator_t<Rng>
    partition_point(Rng& rng, Pred pred, Proj proj = Proj{});

Requires: ForwardIterator’s value type shall be convertible to Predicate’s argument type. [first, last) shall be partitioned by pred and proj, i.e. all elements that satisfy pred shall appear before those that do not there should be an iterator mid such that all_of(first, mid, pred, proj) and none_of(mid, last, pred, proj) are both true.

Returns: An iterator mid such that all_of(first, mid, pred, proj) and none_of(mid, last, pred, proj) are both true.

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

### 25.4 Sorting and related operations

All the operations in 25.4 have two versions: one that takes a function object of type Compare and one that uses an operator. A function object of type Compare defaults to less\. Compare Comp is a function object type [20.9] a callable object [20.9.2]. The return value of the function call Comp applied to an object of type Compare Comp, when contextually converted to bool (Clause 4), yields true if the first argument of the call is less than the second, and false otherwise. Compare Comp comp is used throughout for algorithms assuming an ordering relation. It is assumed that comp will not apply any non-constant function through the dereferenced iterator.

For all algorithms that take Compare, there is a version that uses operator instead. That is, comp(*i, *j) != false defaults to *i < *j != false. For algorithms other than those described in 25.4.3 to work correctly, comp has to induce a strict weak ordering on the values.

[Editor’s note: REVIEW: The above (struck) sentence implies that the binary search algorithms do not require a strict weak ordering relation, but the “Palo Alto report” is clear that they do. Which is it?]

[Editor’s note: The following description of “strict weak order” has moved to the definition of the StrictWeakOrder concept (19.5.6).]

The term strict refers to the requirement of an irreflexive relation (!comp(x, x) for all x), and the term weak to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define equiv(a, b) as !comp(a, b) && !comp(b, a), then the requirements are that comp and equiv both be transitive relations:

\[
\begin{align*}
\text{(4.1)} & \quad \text{comp(a, b) \&\& comp(b, c) implies comp(a, c)} \\
\text{(4.2)} & \quad \text{equiv(a, b) \&\& equiv(b, c) implies equiv(a, c) [Note: Under these conditions, it can be shown that} \\
\text{(4.2.1)} & \quad \text{equiv is an equivalence relation} \\
\text{(4.2.2)} & \quad \text{comp induces a well-defined relation on the equivalence classes determined by equiv} \\
\text{(4.2.3)} & \quad \text{The induced relation is a strict total ordering. -- end note]}
\end{align*}
\]
A sequence is sorted with respect to a comparator and projection \( \text{comp} \) and \( \text{proj} \) if for every iterator \( i \) pointing to the sequence and every non-negative integer \( n \) such that \( i + n \) is a valid iterator pointing to an element of the sequence, \( \text{comp}(*\!(i + n), *i) \Rightarrow false \) \( \text{INVOLVE} \)(\( \text{comp}, \text{INVOLVE} \)(\( \text{proj}, *(i + n) \)), \( \text{INVOLVE} \)(\( \text{proj}, *i) \)) \Rightarrow false.

A sequence \([\text{start}, \text{finish})\) is 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 \( a \) and \( b \) are considered equivalent if and only if \( ! (a < b) \&\& ! (b < a) \).

### 25.4.1 Sorting [alg.sort]

#### 25.4.1.1 \texttt{sort} [sort]

```cpp
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

```cpp
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{});
```

```cpp
template<RandomAccessRange Rng, class Comp = less, class Proj = identity>
requires Sortable<IteratorType iterator_t<Rng>, Comp, Proj>()
IteratorType safe_iterator_t<Rng>
sort(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1. **Effects:** Sorts the elements in the range \([\text{first}, \text{last})\).
2. **Requires:** \texttt{RandomAccessIterator} shall satisfy the requirements of \texttt{ValueSwappable} (19.2.10). The type of \texttt{*first} shall satisfy the requirements of \texttt{MoveConstructible} (Table 20) and of \texttt{MoveAssignable} (Table 22).
3. **Complexity:** \( O(N \log(N)) \) (where \( N = \text{last} - \text{first} \)) comparisons.

#### 25.4.1.2 \texttt{stable_sort} [stable.sort]

```cpp
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

```cpp
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{});
```

```cpp
template<RandomAccessRange Rng, class Comp = less, class Proj = identity>
requires Sortable<IteratorType iterator_t<Rng>, Comp, Proj>()
IteratorType safe_iterator_t<Rng>
```

§ 25.4.1.2
stable_sort(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

Effects: Sorts the elements in the range \([\text{first}, \text{last})\).

Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (19.2.10). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

Complexity: It does at most \(N \log^2(N)\) (where \(N = \text{last} - \text{first}\)) comparisons; if enough extra memory is available, it is \(N \log(N)\).

Remarks: Stable (17.6.5.7).

25.4.1.3 partial_sort

```cpp
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                 RandomAccessIterator middle,
                 RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first,
                 RandomAccessIterator middle,
                 RandomAccessIterator last,
                 Compare comp);
```

```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<IteratorType iterator_t<Rng>, Comp, Proj>()
IteratorType safe_iterator_t<Rng>
partial_sort(Rng& rng, IteratorType iterator_t<Rng> middle, Comp comp = Comp{},
            Proj proj = Proj{});
```

Effects: Places the first \(\text{middle} - \text{first}\) sorted elements from the range \([\text{first}, \text{last})\) into the range \([\text{first}, \text{middle})\). The rest of the elements in the range \([\text{middle}, \text{last})\) are placed in an unspecified order.

Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (19.2.10). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable (Table 22).

Complexity: It takes approximately \((\text{last} - \text{first}) * \log(\text{middle} - \text{first})\) comparisons.

25.4.1.4 partial_sort_copy

```cpp
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                 RandomAccessIterator result_first,
                 RandomAccessIterator result_last);
```

```cpp
template<class InputIterator, class RandomAccessIterator, class Compare>
partial_sort_copy(InputIterator first, InputIterator last,
                 RandomAccessIterator result_first,
                 RandomAccessIterator result_last,
                 Compare comp);
```
```
RandomAccessIterator result_last,
Compare comp);

template<InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
class RComp = less<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyCopyable<I1, I2> & & Sortable<I2, Comp, Proj2>() & &
IndirectCallableStrictWeakOrder<Comp, Eprojected<I1, Proj1>, Eprojected<I2, Proj2>>()
I2
partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, RandomAccessRange Rng2, class RComp = less<>,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyCopyable<IteratorType_iterator_t<Rng1>, IteratorType_iterator_t<Rng2>>() & &
Sortable<IteratorType_iterator_t<Rng2>, Comp, Proj2>() & &
IndirectCallableStrictWeakOrder<Comp, Eprojected<IteratorType_iterator_t<Rng1>, Proj1>,
    Eprojected<IteratorType_iterator_t<Rng2>, Proj2>>()
IteratorType_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).

Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (19.2.10). The
type of *result_first shall satisfy the requirements of MoveConstructible (Table 20) and of Move-
Assignable (Table 22).

Complexity: Approximately (last - first) * log(min(last - first, result_last - result_first -
first)) comparisons.

25.4.1.5 is_sorted [is.sorted]

template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableStrictWeakOrder<Eprojected<I, Proj>> Comp = less<>>
bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<Eprojected<IteratorType_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

1

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last, Compare comp);

Returns: is_sorted_until(first, last, comp, proj) == last

2

template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last, Compare)

§ 25.4.1.5
```
template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallableStrictWeakOrder<Projected<I>, Proj>> Comp = less<>>
I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
IndirectCallableStrictWeakOrder<Projected<IteratorType_t<Rng>, Proj>> Comp = less<>>
IteratorType_saf_iter_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.

25.4.2 Nth element

template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);

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<IteratorType_t<Rng>, Comp, Proj>()
IteratorType_saf_iter_t<Rng>
nth_element(Rng&& rng, IteratorType_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});

1 After nth_element the element in the position pointed to by nth is the element that would be
in that position if the whole range were sorted, unless nth == last. Also for every iterator i in
the range [first, nth) and every iterator j in the range [nth, last) it holds that:
(*j < *i) or
\text{INVOLVE}(\text{comp}, \text{INVOLVE}(\text{proj}, *j), \text{INVOLVE}(\text{proj}, *i)) == \text{false}.

2 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (19.2.10). The
type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable
(Table 22).

Complexity: Linear on average.

25.4.3 Binary search

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 implied
or explicit 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.

25.4.3.1 lower_bound

[lower_bound]
template<class ForwardIterator, class T>
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);

template<ForwardIterator I, Sentinel<I> S, TotallyOrdered class T, class Proj = identity,
IndirectCallableStrictWeakOrder<const T*, Projected<I, Proj>> Comp = less<>>
I
lower_bound(I first, S last, const T& value, Comp comp = Comp{},
Proj proj = Proj{});

template<ForwardRange Rng, TotallyOrdered class T, class Proj = identity,
IndirectCallableStrictWeakOrder<const T*, Projected<IteratorType iterator_t<Rng>, Proj>> Comp = less<>>
IteratorType safe_iterator_t<Rng>
lower_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

1 ReQUIRES: The elements e of [first, last) shall be partitioned with respect to the expression e < value ? \text{INVoke}(comp, \text{INVoke}(proj, e), value) : \text{comp}(e, value).

2 Returns: The furthermost iterator i in the range [first, last] such that for every iterator j in the range [first,i) the following corresponding conditions hold: \text{\texttt{!value < *j or \text{comp(*j, value)) != false}}}.

3 Complexity: At most $\log_2(last - first) + O(1)$ comparisons and projection.

25.4.3.2 upper_bound [upper.bound]

template<class ForwardIterator, class T>
ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);

template<ForwardIterator I, Sentinel<I> S, TotallyOrdered class T, class Proj = identity,
IndirectCallableStrictWeakOrder<const T*, Projected<I, Proj>> Comp = less<>>
I
upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, TotallyOrdered class T, class Proj = identity,
IndirectCallableStrictWeakOrder<const T*, Projected<IteratorType iterator_t<Rng>, Proj>> Comp = less<>>
IteratorType safe_iterator_t<Rng>
upper_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

1 ReQUIRES: The elements e of [first, last) shall be partitioned with respect to the expression \text{!(value < e) ? \text{INVoke}(comp, value, \text{INVoke}(proj, e)) : \text{comp}(value, e)}.

2 Returns: The furthermost iterator i in the range [first, last] such that for every iterator j in the range [first,i) the following corresponding conditions hold: \text{\texttt{!value < *j or \text{comp(value, *j)}}}.
\begin{itemize}
\item \texttt{\texttt{false} \texttt{INVOLVE}(comp, value, \texttt{INVOLVE}(proj, *j)) = \texttt{false}}.
\end{itemize}

\textbf{Complexity:} At most \( \log_2(\text{last - first}) + O(1) \) comparisons and applications of the comparison function and projection.

\subsection*{25.4.3.3 equal\_range}

\begin{verbatim}
25.4.3.3 equal\_range

template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

template<ForwardIterator I, Sentinel<I> S, TotallyOrdered class T, class Proj = identity,
IndirectCallableStrictWeakOrder<const T*, Projected<I, Proj>> Comp = less<>>
tagged_pair<tag::begin(I), tag::end(I)>
equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, TotallyOrdered class T, class Proj = identity,
IndirectCallableStrictWeakOrder<const T*, Projected<IteratorType iterator_t<Rng>, Proj>> Comp = less<>>
tagged_pair<tag::begin(IteratorType safe_iterator_t<Rng>), tag::end(IteratorType safe_iterator_t<Rng>)>
equal_range(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
\end{verbatim}

\textbf{Requires:} The elements \( e \) of \([first, last)\) shall be partitioned with respect to the expressions \( e < value \texttt{INVOLVE}(comp, \texttt{INVOLVE}(proj, e), value) \) and \(!((value < e) \texttt{INVOLVE}(comp, value, \texttt{INVOLVE}(proj, e)) \text{ or } \text{comp}(e, value) \text{ and } \neg\text{comp}(value, e)) \). Also, for all elements \( e \) of \([first, last)\), \( e < value \texttt{INVOLVE}(comp, \texttt{INVOLVE}(proj, e), value) \) shall imply \(!((value < e) \texttt{INVOLVE}(comp, value, \texttt{INVOLVE}(proj, e)) \text{ or } \text{comp}(e, value) \text{ shall imply } \neg\text{comp}(value, e)) \).

\textbf{Returns:}

\begin{verbatim}
make_pair(lower_bound(first, last, value),
upper_bound(first, last, value))
\end{verbatim}

or

\begin{verbatim}
make_pair(lower_bound(first, last, value, comp, proj),
upper_bound(first, last, value, comp, proj))
\end{verbatim}

\textbf{Complexity:} At most \( 2 + \log_2(\text{last - first}) + O(1) \) comparisons and applications of the comparison function and projection.

\subsection*{25.4.3.4 binary\_search}

\begin{verbatim}
25.4.3.4 binary\_search

template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
\end{verbatim}
template<ForwardIterator I, Sentinel<I> S, TotallyOrdered class T, class Proj = identity, IndirectCallableStrictWeakOrder<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, TotallyOrdered class T, class Proj = identity, IndirectCallableStrictWeakOrder<const T*, projected<IteratorType 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 e<~ value INVOKE(comp, INVOKE(proj, e), value) and !(value < e) INVOKE(comp, value, INVOKE(proj, e)) or comp(e, value) and !comp(value, e). Also, for all elements e of [first, last), e<~ value INVOKE(comp, INVOKE(proj, e), value) shall imply !(value < e) INVOKE(comp, value, INVOKE(proj, e)) or comp(e, value) shall imply !comp(value, e).

2. Returns: true if there is an iterator i in the range [first, last) that satisfies the corresponding conditions: !(i<value) && !(value < i) INVOKE(comp, INVOKE(proj, *i), value) == false && INVOKE(comp, value, INVOKE(proj, *i)) == false || comp(*i, value) == false && comp(value, *i) == false.

3. Complexity: At most \( \log_2(last - first) + \Theta(1) \) comparisons and applications of the comparison function and projection.

25.4.4 Merge [alg.merge]

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator
merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator
merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, Incrementable 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, Incrementable O, class Comp = less<>>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
tagged_tuple<tag::in1(it<IteratorType safe_iterator_t<Rng1>.), tag::in2(it<IteratorType safe_iterator_t<Rng2>.), tag::out(O)>
merge(Rng1& rng1, Rng2& rng2, O result,
Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{};

**Effects:** Copies all the elements of the two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) into the range \([\text{result}, \text{result_last})\), where \(\text{result_last} = \text{result} + (\text{last1} - \text{first1}) + (\text{last2} - \text{first2})\), such that the resulting range satisfies is_sorted(result, result_last, comp).

**Requires:** The ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) shall be sorted with respect to \(\text{comp}, \text{proj1}, \text{and proj2}\). The resulting range shall not overlap with either of the original ranges.

**Returns:** \(\text{result} + (\text{last1} - \text{first1}) + (\text{last2} - \text{first2})\) make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result + (last1 - first1) + (last2 - first2)).

**Complexity:** At most \((\text{last1} - \text{first1}) + (\text{last2} - \text{first2}) - 1\) applications of the comparison function and each projection.

**Remarks:** Stable (17.6.5.7).
template<BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
        class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
    requires Sortable<IteratorType iterator_t<Rng>, Comp, Proj>
    IteratorType safe_iterator_t<Rng>
    inplace_merge(Rng& rng, IteratorType 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 *i < *(i - 1) or, respectively,
comp(*i, *(i - 1)) \text{ invoke } \text{comp, invoke } \text{proj, } *i, \text{ invoke } \text{proj, } *(i - 1))\) will be false.

Requires: The ranges [first,middle) and [middle,last) shall be sorted with respect to operator<
or comp and proj. BidirectionalIterator shall satisfy the requirements of ValueSwappable (19.2.10).
The type of *first shall satisfy the requirements of MoveConstructible (Table 20) and of MoveAssignable
(Table 22).

Returns: last

Complexity: When enough additional memory is available, \((last - first) - 1\) comparisons
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 (17.6.5.7).

25.4.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They also work with
multisets (23.4.7) 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.

25.4.5.1 includes

template<class InputIterator1, class InputIterator2>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  Compare comp);

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
        class Proj1 = identity, class Proj2 = identity,
        IndirectCallableStrictWeakOrder<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
    bool
    includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
              Proj1 pro1j1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, InputRange Rng2, class Proj1 = identity,
        class Proj2 = identity,
        IndirectCallableStrictWeakOrder<Projected<IteratorType iterator_t<Rng1>, Proj1>,

§ 25.4.5.1
includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
   Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

1 Returns: true if [first2,last2) is empty or if every element in the range [first2,last2) is contained in the range [first1,last1). Returns false otherwise.

2 Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons applications of the comparison function and projections.

25.4.5.2 set_union

1 Effects: Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

2 Requires: The resulting range shall not overlap with either of the original ranges.

3 Returns: The end of the constructed range make_tagged_tuple<tag::in1, tag::in2, tag::out>({last1, last2, result + n}), where n is the number of elements in the constructed range.

4 Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons applications of the comparison function and projections.

5 Remarks: If [first1,last1) contains m elements that are equivalent to each other and [first2, last2) contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then max(n – m, 0) elements from the second range shall be copied to the output range, in order.
25.4.5.3 set_intersection

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result, Compare comp);
```

```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>()
O
set_intersection(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_iterator_t<Rng1>, iterator_iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
O
set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 Effects: Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

2 Requires: The resulting range shall not overlap with either of the original ranges.

3 Returns: The end of the constructed range.

4 Complexity: At most \(2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1\) comparisons and applications of the comparison function and projections.

5 Remarks: If \([\text{first1}, \text{last1})\) contains \(m\) elements that are equivalent to each other and \([\text{first2}, \text{last2})\) 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.

25.4.5.4 set_difference

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result, Compare comp);
```
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, 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<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
tagged_pair<tag::in1(iterator_t<Rng1>), tag::out(O)>
  set_difference(Rng1& rng1, Rng2&& rng2, 0 result,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

1 Effects: Copies the elements of the range [first1, last1) which are not present in the range [first2,
2 last2) to the range beginning at result. The elements in the constructed range are sorted.
3 Requires: The resulting range shall not overlap with either of the original ranges.
4 Returns: The end of the constructed range [last1, result + n], where n is the number of elements
5 in the constructed range.
6 Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 comparisons of the comparison function and projections.
7 Remarks: If [first1, last1) contains m elements that are equivalent to each other and [first2,
8 last2) contains n elements that are equivalent to them, the last max(m - n, 0) elements from [first1,
9 last1) shall be copied to the output range.

25.4.5.5 set_symmetric_difference

template<class InputIterator1, class InputIterator2,
  class OutputIterator>
OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2,
  OutputIterator result);

template<class InputIterator1, class InputIterator2,
  class OutputIterator, class Compare>
OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2,
  OutputIterator result, Compare comp);

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
  WeaklyIncrementable 0, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<I1, I2, 0, 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, 0 result,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, InputRange Rng2, WeaklyIncrementable 0,
  class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<IteratorType iterator_t<Rng1>, IteratorType iterator_t<Rng2>, 0, Comp, Proj1, Proj2>()
tagged_tuple<tag::in1(iterator_t<Rng1>),
  tag::in2(iterator_type_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{});

§ 25.4.5.5
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 \([\text{first1}, \text{last1})\) that are not present in the range \([\text{first2}, \text{last2})\), and the elements of the range \([\text{first2}, \text{last2})\) that are not present in the range \([\text{first1}, \text{last1})\) to the range beginning at \text{result}. The elements in the constructed range are sorted.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: The end of the constructed range \text{make_tagged_tuple}\langle \text{tag::in1}, \text{tag::in2}, \text{tag::out}\rangle(\text{last1}, \text{last2}, \text{result} + n), where \(n\) is the number of elements in the constructed range.

Complexity: At most \(2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1\) comparisons applications of the comparison function and projections.

Remarks: If \([\text{first1}, \text{last1})\) contains \(m\) elements that are equivalent to each other and \([\text{first2}, \text{last2})\) contains \(n\) elements that are equivalent to them, then \(|m - n|\) of those elements shall be copied to the output range: the last \(m - n\) of these elements from \([\text{first1}, \text{last1})\) if \(m > n\), and the last \(n - m\) of these elements from \([\text{first2}, \text{last2})\) if \(m < n\).

### 25.4.6 Heap operations [alg.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.

\text{make_heap}() converts a range into a heap and \text{sort_heap}() turns a heap into a sorted sequence.

#### 25.4.6.1 push_heap [push.heap]

\begin{verbatim}
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
          class Proj = identity>
  requires Sortable<I, Comp = less<>>,
          Proj Proj = identity>
  I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessRange Rng, class Comp = less<>,
         class Proj = identity>
  requires Sortable<IteratorType iterator_t<Rng>, Comp, Proj = identity>
  iterator_t<Rng> push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
\end{verbatim}

Effects: Places the value in the location \text{last} - 1 into the resulting heap \([\text{first}, \text{last})\).

Requires: The range \([\text{first}, \text{last} - 1)\) shall be a valid heap. The type of \(*\text{first}\) shall satisfy the MoveConstructible requirements (Table 20) and the MoveAssignable requirements (Table 22).

Returns: \text{last}

Complexity: At most \(\log(\text{last} - \text{first})\) comparisons applications of the comparison function and projection.
25.4.6.2 pop_heap

```cpp
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

```cpp
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{});
```

```cpp
template<RandomAccessRange Rng, class Comp = less<>,
class Proj = identity>
requires Sortable<IteratorType iterator_t<Rng>, Comp, Proj>()
IteratorType safe_iterator_t<Rng>
pop_heap(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 Requires: The range \([\text{first}, \text{last})\) shall be a valid non-empty heap. \text{RandomAccessIterator} shall satisfy the requirements of \text{ValueSwappable} (19.2.10). The type of \*\text{first} shall satisfy the requirements of \text{MoveConstructible} (Table 20) and of \text{MoveAssignable} (Table 22).

2 Effects: Swaps the value in the location \text{first} with the value in the location \text{last} - 1 and makes \([\text{first}, \text{last} - 1)\) into a heap.

3 Returns: last

4 Complexity: At most \(2 \times \log(\text{last} - \text{first})\) comparisons applications of the comparison function and projection.

25.4.6.3 make_heap

```cpp
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

```cpp
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{});
```

```cpp
template<RandomAccessRange Rng, class Comp = less<>,
class Proj = identity>
requires Sortable<IteratorType iterator_t<Rng>, Comp, Proj>()
IteratorType safe_iterator_t<Rng>
make_heap(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 Requires: The type of \*\text{first} shall satisfy the \text{MoveConstructible} requirements (Table 20) and the \text{MoveAssignable} requirements (Table 22).

2 Effects: Constructs a heap out of the range \([\text{first}, \text{last})\).

3 Returns: last

4 Complexity: At most \(3 \times (\text{last} - \text{first})\) comparisons applications of the comparison function and projection.

§ 25.4.6.3 179
25.4.6.4 sort_heap

```cpp
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
class Proj = identity>
requires Sortable<I, Comp, Proj>(1)
I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
requires Sortable<IteratorType_iterator_t<Rng>, Comp, Proj>()
safe_iterator_t<Rng> sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1. **Effects:** Sorts elements in the heap \([\text{first}, \text{last})\).
2. **Requires:** The range \([\text{first}, \text{last})\) shall be a valid heap. \texttt{RandomAccessIterator} shall satisfy the requirements of \texttt{ValueSwappable} (19.2.10). The type of \(*\text{first}\) shall satisfy the requirements of \texttt{MoveConstructible} (Table 20) and of \texttt{MoveAssignable} (Table 22).
3. **Returns:** last
4. **Complexity:** At most \(N \log(N)\) comparisons (where \(N = \text{last} - \text{first}\)).

25.4.6.5 is_heap

```cpp
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallableStrictWeakOrder<Projected<I, Proj>> Comp = less<>>
bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessRange Rng, class Proj = identity,
IndirectCallableStrictWeakOrder<Projected<IteratorType_iterator_t<Rng>, Proj>> Comp = less<>>
bool is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1. **Returns:** \texttt{is_heap_until(first, last, comp, proj) == last}

§ 25.4.6.5
template<RandomAccessRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<Projected<IteratorType iterator_t<Rng>, Proj>> Comp = less<>
    IteratorType 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.

25.4.7 Minimum and maximum

[alg.min.max]

template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare> 
    constexpr const T& min(const T& a, const T& b, Compare comp);

template<TotallyOrdered_T> 
    constexpr const T& min(const T& a, const T& b);

template<class T, class Proj = identity, 
    IndirectCallableStrictWeakOrder<Projected<const T*, Proj>> Comp = less<>
    requires StrictWeakOrder<Comp, T> 
    constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

1 Requires: Type T is LessThanComparable (Table 18).
2 Returns: The smaller value.
3 Remarks: Returns the first argument when the arguments are equivalent.

template<class T> 
    constexpr T min(initializer_list<T> t);
template<class T, class Compare> 
    constexpr T min(initializer_list<T> t, Compare comp);

template<TotallyOrdered_T> 
    requires Semiregular<T> 
    constexpr T min(initializer_list<T> t, Compare comp);

template<InputRange Rng> 
    requires TotallyOrdered<ValueType<IteratorType<Rng>>> && 
    Semiregular<ValueType<IteratorType<Rng>>> 
    ValueType<IteratorType<Rng>> min(Rng& rng);

template<SemiregularCopyable T, class Proj = identity, 
    IndirectCallableStrictWeakOrder<Projected<value_type_t<IteratorType iterator_t<Rng>, Proj>>, T> 
    requires StrictWeakOrder<Comp, T> 
    constexpr T min(initializer_list<T> t, Compare comp = Comp{}, Proj proj = Proj{});

4 Requires: T is LessThanComparable and CopyConstructible and t.size() > 0, \( \text{distance(begin(rng), end(rng))} > 0 \).
5 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.

```cpp
template<class T> constexpr const T& max(const T& a, const T& b);
template<class T, class Compare>
constexpr const T& max(const T& a, const T& b, Compare comp);
template<TotallyOrdered T>
constexpr T max(const T& a, const T& b);
```

Requires: Type T is LessThanComparable (Table 18).

Returns: The larger value.

Remarks: Returns the first argument when the arguments are equivalent.

```cpp
template<class T>
constexpr T max(initializer_list<T> t);
template<class T, class Compare>
constexpr T max(initializer_list<T> t, Compare comp);
template<TotallyOrdered T>
requires Semiregular<T>
constexpr T max(initializer_list<T> t, Compare comp);
```

Requires: T is LessThanComparable and CopyConstructible and t.size() > 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.

```cpp
template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```
template<typename T, class Proj = identity, 
IndirectCallableStrictWeakOrder<Projected<const T*>, Proj>> Comp = less<>>
requires StrictWeakOrder<FunctionType<Comp>, T>
constexpr tagged_pair<tag::min(const T&), tag::max(const T&)> 
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

Requires: Type T shall be LessThanComparable (Table 18).

Returns: \texttt{pair<const T&, const T&}(\{b, a\}) if \(b\) is smaller than \(a\), and \texttt{pair<const T&, const T&}(\{a, b\}) otherwise.

Remarks: Returns \texttt{pair<const T&, const T&}(\{a, b\}) when the arguments are equivalent.

Complexity: Exactly one comparison and exactly two applications of the projection.

template<

    template<typename T>
    constexpr pair<T, T> minmax(initializer_list<T> t);

    template<typename T, class Compare>
    constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

    template<typename T, class Proj = identity, 
IndirectCallableStrictWeakOrder<Projected<const T*, Proj>> Comp = less<>>
requires StrictWeakOrder<FunctionType<Comp>, T>
constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(initializer_list<T> t, Compare comp = Comp{}, Proj proj = Proj{});

template<typename InputRange Rng, class Proj = identity, 
IndirectCallableStrictWeakOrder<Projected<IteratorType<Rng>, Proj> t>, Proj>> Comp = less<>>
requires SemiregularCopyable<
    ValueType<
        IteratorType<
            iterator_t<Rng>, Proj>
    >, Proj>>
    tagged_pair<tag::min(ValueType<
        IteratorType<
            iterator_t<Rng> >>), tag::max(ValueType<
        IteratorType<
            iterator_t<Rng> >>))
    minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Requires: T is LessThanComparable and CopyConstructible and \(t\).size() > 0.

Returns: \texttt{pair<T, T>(\{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 \((3/2) \times t\.size() \times (3/2) \times distance(begin(rng), end(rng))\) applications of the corresponding predicate, and at most twice as many applications of the projection.
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<I, Proj>> Comp = less<>>
I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<IteratorType iterator_t<Rng>, Proj>> Comp = less<>>
IteratorType 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 conditions hold: \(((\text{proj}(i) < \text{proj}(j)) \text{ or } \text{comp}(*i, *j) == false)\). Returns last if first == last.

Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparison function and exactly twice as many applications of the projection.

template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<I, Proj>> Comp = less<>>
I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardRange Rng, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<IteratorType iterator_t<Rng>, Proj>> Comp = less<>>
IteratorType 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 conditions hold: \(((\text{proj}(i) > \text{proj}(j)) \text{ or } \text{comp}(*i, *j) == false)\). Returns last if first == last.

Complexity: Exactly max((last - first) - 1, 0) applications of the corresponding comparison function and exactly twice as many applications of the projection.

template<class ForwardIterator>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
IndirectCallableStrictWeakOrder<P projected<I, Proj>> Comp = less<>>
tagged_pair<tag::min(I), tag::max(I)>
minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
template<ForwardRange Rng, class Proj = identity, 
    IndirectCallableStrictWeakOrder<PProjected<IteratorTypeiterator_t<Rng>>, Proj>, Comp = less>>
tagged_pair<tag::min(IteratorTypesafe_iterator_t<Rng>),
tag::max(IteratorTypesafe_iterator_t<Rng>)>
minmax_element(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

Returns: make_pair(first, first) if [first, last) is empty, otherwise make_pair(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 \(\frac{1}{2}(N - 1)\) applications of the corresponding predicate comparison function and at most twice as many applications of the projection, where \(N\) is \(\text{distance}(\text{first}, \text{last})\).

25.4.8 Lexicographical comparison

```
template<class InputIterator1, class Compare>
bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1, 
                            InputIterator2 first2, InputIterator2 last2, 
                            Compare comp);
```

Returns: \(true\) if the sequence of elements defined by the range \([\text{first1}, \text{last1})\) is lexicographically less than the sequence of elements defined by the range \([\text{first2}, \text{last2})\) and \(false\) otherwise.

Complexity: At most \(2 \times \min((\text{last1} - \text{first1}), (\text{last2} - \text{first2}))\) applications of the corresponding comparison \textbf{and projection}.

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.

```c
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
    if (*first1 < *first2) return true;
    if (*first2 < *first1) return false;
}
```
return first1 == last1 && first2 != last2;

using namespace placeholders;
auto&& cmp1 = bind(comp, bind(proj1, _1), bind(proj2, _2));
auto&& cmp2 = bind(comp, bind(proj2, _1), bind(proj1, _2));
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
    if (cmp1(*first1, *first2)) return true;
    if (cmp2(*first2, *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.

25.4.9 Permutation generators

template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                     BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                     BidirectionalIterator last, Compare comp);

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<IteratorType 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 operator< or 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.

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (19.2.10).

Complexity: At most \((last - first)/2\) swaps.

template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);

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<IteratorType 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 operator< or comp and 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.

Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (19.2.10).

Complexity: At most (last - first)/2 swaps.

25.5 C library algorithms

Table 8 describes some of the contents of the header <cstdlib>.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>bsearch qsort</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header <stdlib.h> with the following exceptions:

The function signature:

bsearch(const void *, const void *, size_t, size_t,  
int (*)(const void *, const void *));

is replaced by the two declarations:

extern "C" void* bsearch(const void* key, const void* base,  
size_t nmemb, size_t size,  
int (*compar)(const void*, const void*));
extern "C++" void* bsearch(const void* key, const void* base,  
size_t nmemb, size_t size,  
int (*compar)(const void*, const void*));

both of which have the same behavior as the original declaration.

The function signature:

qsort(void *, size_t, size_t,  
int (*)(const void *, const void *));

is replaced by the two declarations:

extern "C" void qsort(void* base, size_t nmemb, size_t size,  
int (*compar)(const void*, const void*));
extern "C++" void qsort(void* base, size_t nmemb, size_t size,  
int (*compar)(const void*, const void*));

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by base are of trivial type.
[Note: Because the function argument `compar()` may throw an exception, `bsearch()` and `qsort()` are allowed to propagate the exception (17.6.5.12). — end note]

See also: ISO C 7.10.5.
26  Numerics library

Header <experimental/ranges_v1/random> synopsis

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {

    template <class G>
    concept bool UniformRandomNumberGenerator() { return see below; }
}
}}}
```

26.5  Random number generation

26.5.1  Requirements

26.5.1.3  Uniform random number generator requirements

// defined in <experimental/ranges_v1/random>

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {

    template <class G>
    concept bool UniformRandomNumberGenerator() {=
        return requires(G g) {
            typename ResultType<G>;
            requires UnsignedIntegral<ResultType<G> result_of_t<G&>()>; // not required to be equality preserving
            { g() } -> Same<ResultType<G> result_of_t<G&>()>;
            { G::min() } -> Same<ResultType<G> result_of_t<G&>()>;
            { G::max() } -> Same<ResultType<G> result_of_t<G&>()>;
        }
    }
}}}
```

1  A uniform random number generator \( g \) of type \( G \) is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [Note: The degree to which \( g \)'s results approximate the ideal is often determined statistically. — end note]

[Editor’s note: Remove para 2 and 3 and Table 116 (Uniform random number generator requirements).]

2  Let \( g \) be any object of type \( G \). Then type \( G \) models \( \text{UniformRandomNumberGenerator<}G\text{>()} \) is satisfied if and only if

1. Both \( G\text{::min()} \) and \( G\text{::max()} \) are constant expressions (5.19).
2. \( \langle G\text{::min()} < G\text{::max()} \rangle \) is false.
3. \( G\text{::min()} \leq g() < G\text{::max()} \) is false.
4. \( g() \leq G\text{::max()} \).
5. \( g() \) has amortized constant complexity.
Annex D  (normative)  
Compatibility features  
[depr]

1 This Clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

2 These are deprecated features, where deprecated is defined as: Normative for the current edition of the Standard, but having been identified as a candidate for removal from future revisions. An implementation may declare library names and entities described in this section with the deprecated attribute (7.6.5).

D.10 Range-and-a-half algorithms  
[depr.algo.range-and-a-half]

1 The following algorithms are deemed unsafe and are deprecated in this document.

// 25.2.10, mismatch
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,  
class Proj1 = identity, class Proj2 = identity,  
IndirectCallablePredicate<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>  
tagged_pair<tag::in1(I1), tag::in2(I2)>  
mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{},  
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, WeakInputIterator I2,  
class Proj1 = identity, class Proj2 = identity,  
IndirectCallablePredicate<projected<iterator_t<Rng1>, Proj1>,  
projected<I2, Proj2>> Pred = equal_to<>  
tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(I2)>  
mismatch(Rng1&& rng1, I2 first2, Pred pred = Pred{},  
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 25.2.11, equal
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,  
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, Pred pred = Pred{},  
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng1, WeakInputIterator I2, class Pred = equal_to<>,  
class Proj1 = identity, class Proj2 = identity>  
requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()  
bool equal(Rng1&& rng1, I2 first2, Pred pred = Pred{},  
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 25.2.12, is_permutation
template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,  
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,  
Pred pred = Pred{},  
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
template<ForwardRange Rng1, ForwardIterator I2, class Pred = equal_to<>,
        class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool is_permutation(Rng1&& rng1, I2 first2, Pred pred = Pred{},
                        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 25.3.3, swap_ranges
template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in1(I1), tag::in2(I2)>
    swap_ranges(I1 first1, S1 last1, I2 first2);

template<ForwardRange Rng, ForwardIterator I>
requires IndirectlySwappable<iterator_t<Rng>, I>()
tagged_pair<tag::in1(safe_iterator_t<Rng>), tag::in2(I)>
    swap_ranges(Rng&& rng1, I first2);

// 25.3.4, transform
template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2, WeaklyIncrementable O,
            class 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, O result,
              F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputRange Rng, WeakInputIterator I, WeaklyIncrementable O, class F,
            class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng>, Proj1>, projected<I, Proj2>)>>(
tagged_tuple<tag::in1(safe_iterator_t<Rng>), tag::in2(I), tag::out(O)>
    transform(Rng&& rng1, I first2, O result,
              F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
Annex A  (informative)
Acknowledgements

The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as “The Palo Alto” report (1.2), which was developed by a large group of experts as a test of the expressive power of the idea of concepts.

I would like to thank Andrew Sutton for his work on the Concepts TS (1.2), for his help formalizing the ideas of the range-v3 library [2] on which this work is based, and for his review of this document.

Sean Parent has made major contributions to both the foundations and the wording of this paper.

Stephan T. Lavavej offered a careful review of much of this document, a non-trivial undertaking.

I would also like to thank the members of the Ranges SG who offered feedback on early drafts; especially, Tony Van Eerd, Casey Carter, and Walter Brown.

This work was made possible by a generous grant from the Standard C++ Foundation.
Annex B  (informative)
Compatibility  [diff]

B.1  C++ and Ranges  [diff.cpp]
1 This section details the known breaking changes likely to effect user code when being ported to the version
of the Standard Library described in this document.

B.1.1  Algorithm Return Types  [diff.cpp.algo_return]
1 The algorithms described in this document permit the type of the end sentinel to differ from the type of the
begin iterator. This is so that the algorithms can operate on ranges for which the physical end position is
not yet known.

2 The physical end position of the input range is determined during the execution of many of the algorithms. Rather
than lose that potentially useful information, the design presented here has such algorithms return
the iterator position of the end of the range. In many cases, this is a breaking change. Some algorithms
that return iterators in today’s STL are changed to return pairs, and algorithms that return pairs today are
changed to return tuples. This is likely to be the most noticeable breaking change.

3 Alternate designs that were less impactful were considered and dismissed. See Section 3.3.6 in N4128 (1.2)
for a discussion of the issues.

B.1.2  Stronger Constraints  [diff.cpp.constraints]
1 In this proposal, many algorithms and utilities get stricter type checking. For example, algorithms con-
strained with LessThanComparable today are constrained by StrictTotallyOrdered in this document.
This concept requires types to provide all the relational operators, not just operator<.

2 The use of coarser-grained, higher-level concepts in algorithm constraints is to make the type checks more
semantic in nature and less syntactic. It also has the benefit of being less burdensome while giving algorithm
implementors greater implementation freedom. This approach is in contrast to the previous effort to add
concepts to the Standard Library in the C++0x timeframe, which saw a proliferation of small, purely
syntactic concepts and algorithm constraints that merely restated the algorithms’ implementation details
more verbosely in the algorithms’ function signatures.

3 The potential for breakage must be carefully weighed against the integrity and complexity of the constraints
system. The coarseness of the concepts may need to change in response to real-world usage.

B.1.3  Constrained Functional Objects  [diff.cpp.functional]
1 The algorithm design described in this document assumes that the function objects std::equal_to and
std::less get constraints added to their function call operators. (The former is constrained with Equality-
Comparable and the latter with StrictTotallyOrdered). Similar constraints are added to the other func-
tion objects in <functional>. Also, the so-called transparent relational function objects (less<void> and
friends) have their return type coerced to bool. As with the coarsely-grained algorithm constraints, these
function object constraints are likely to cause user code to break.

2 Real-world experience is needed to assess the seriousness of the breakage. From a correctness point of view,
the constraints are logical and valuable, but it’s possible that for the sake of compatibility we provide both
constrained and unconstrained functional objects.
B.1.4 Iterators and Default-Constructibility [diff.cpp.defaultconstruct]

1 In today’s STL, iterators need not be default-constructible. The `Iterator` concept described in this document requires default-constructibility. This could potentially cause breakage in users’ code. Also, it makes the implementation of some types of iterators more complicated. Any iterator that has members that are not default constructible (e.g., an iterator that contains a lambda that has captured by reference) must take special steps to provide default-constructibility (e.g., by wrapping non-default-constructible types in `std::optional`). This can weaken class invariants.

2 The guarantee of default-constructibility simplifies the implementation of much iterator- and range-based code that would otherwise need to wrap iterators in `std::optional`. But the needs of backward-compatibility, the extra complexity to iterator implementors, and the weakened invariants may prove to be too great a burden.

3 We may in fact go even farther and remove the requirement of default-constructibility from the `Semiregular` concept. Time and experience will give us guidance here.

B.1.5 `iterator_traits` cannot be specialized [diff.cpp.iteratortraits]

1 In this STL design, `iterator_traits` changes from being a class template to being an `template alias` template. This is intentionally break any code that tries to specialize it. In its place are the three class templates `difference_type`, `value_type`, and `iterator_category`. The need for this traits balkanization is because the associated types belong to separate concepts: `difference_type` belongs to `WeaklyIncrementable`; `value_type` belongs to `Readable`; and `iterator_category` belongs to `WeakInputIterator`.

2 This breakage is intentional and inherent in the decomposition of the iterator concepts established by The Palo Alto report (1.2).

B.2 Ranges and the Palo Alto TR (N3351) [diff.n3351]

1 The Palo Alto report (1.2) presents a comprehensive design for the Standard Template Library constrained with concepts. It served both as a basis for the Concepts Lite language feature and for this document. However, this document diverges from the Palo Alto report in small ways. The differences are in the interests of backwards compatability, to avoid confusing a large installed base of programmers already familiar with the STL, and to keep the scope of this document as small as possible. This section describes the ways in which the two suggested designs differ.

B.2.1 Sentinels [diff.n3351.sentinels]

1 In the design presented in this document, the type of a range’s end delimiter may differ from the iterator representing the range’s start position. The reasons for this change are discribed in N4128 (1.2). This causes a number of difference from the Palo Alto report:

(1.1) — The algorithms get an additional constraint for the sentinel.

(1.2) — The return types of the algorithms are changed as described above (B.1.1).

(1.3) — Some algorithms have operational semantics that require them to know the physical end position (e.g., `reverse`). Those algorithms must make an \(O(N)\) probe for the end position before proceeding. This does not change the operational semantics of any code that is valid today (the probe is unnecessary when the types of the begin and end are the same), and even when the probe is needed, in no cases does this change the complexity guarantee of any algorithm.

B.2.2 Callables and Projections [diff.n3351.invok_proj]

1 Adobe’s Source Libraries [1] pioneered the use of `callables` and `projections` in the standard algorithms. Callables let users pass member pointers where the algorithms expect callables, saving users the trouble of using a binder or a lambda. Projections are extra optional arguments that give users a way to trivially
transform input data on the fly during the execution of the algorithms. Neither significantly changes the operational semantics of the algorithms, but they do change the form of the algorithm constraints. To deal with the extra complexity of the constraints, the design presented here adds higher-level composite concepts for concisely expressing the necessary relationships between callables, iterators, and projections.

B.2.3 No Distinct DistanceType Associated Type [diff.n3351.distance_type]

In the Palo Alto report, the `WeaklyIncrementable` concept has an associated type called `DistanceType`, and the `RandomAccessIterator` concepts adds another associated type called `DifferenceType`. The latter is required to be convertible to the former, but they are not required to be the same type. (`DifferenceType` is required to be a signed integral type, but `DistanceType` need not be signed.) Although sensible from a soundness point of view, the author of this paper feels this is potentially a rich source of confusion. This paper hews closer to the current standard by having only one associated type, `DifferenceType`, and requiring it to be signed.

B.2.4 Distance Primitive is O(1) for Random Access Iterators [diff.n3351.distance_algo]

In the Palo Alto report, the `distance` iterator primitive for computing the distance from one iterator position to another is not implemented in terms of `operator-` for random access iterators. `distance`, according to the report, should always be $\mathcal{O}(N)$. It reads:

```
    The standard mandates a different definition for random access iterators: distance(i, j) == j - i. We see this as a specification error; the guarantees of the distance operation have been weakened for an iterator specialization.
```

In our design, we consider the two operations to be distinct.

The design presented in this document keeps the specialization for random access iterators. To do otherwise would be to silently break complexity guarantees in an unknown amount of working code.

To address the concern about weakened guarantees of the `distance` primitive, the design presented here requires that random access iterators model `SizedIteratorRange` (24.5.1). The `SizedIteratorRange` concept requires that $b - a$ return the number of times $a$ would have to be incremented to make it compare equal to $b$. Any type purporting to be a random access iterator that fails to meet that requirement is by definition not a valid random access iterator.

B.2.5 Output Iterators [diff.n3351.output_iters]

The Palo Alto report does not define concepts for output iterators, making do with `WeaklyIncrementable`, `Writable`, and (where needed) `EqualityComparable`. The author of this document sees little downside to grouping these into the familiar `OutputIterator` concept and the more frequently-needed `WeakOutputIterator`. Even if they are not strictly needed, their absence would be surprising.

B.2.6 No Algorithm Reformulations [diff.n3351.no_eop_algos]

Between the standardization of the Standard Library and the Palo Alto report, much new research was done to further generalize the standard algorithms (see “Element of Programming”, Stepanov, McJones [4]). The algorithms presented in The Palo Alto report reflect the results of that research in the algorithm constraints, some of which (e.g., `sort`, `inplace_merge`) take iterators with weaker categories than they do in the current standard. The design presented in this document does not reflect those changes. Although those changes are desirable, generalizing the algorithms as described in The Palo Alto report feels like it would be best done in a separate proposal.
Annex C  (informative)

Future Work

This document brings ranges and concepts to a minimal useful subset of the Standard Library. A proper and full implementation has many more moving parts. In addition, we can use the opportunity this work presents to address long-standing shortcomings in the Standard Library. Some of these future work items are described below.

C.1 Proxy Iterators

As early at 1998 when Herb Sutter published his “When is a Container not a Container” [5] article, it was known that proxy iterators were a challenge that the current iterator concept hierarchy could not meet. The problem stems from the fact that the ForwardIterator concept as specified in the current standard requires the iterator’s reference type to be a true reference, not a proxy. The Palo Alto report lifts this restriction in its respecification of the iterator concepts but doesn’t actually solve the problem. The majority of algorithms, once you study the constraints, do not accept many interesting proxy iterator types.

The author of this document has researched a possible library-only solution to the problem and implemented it in the range-v3 library. (The details of that solution are described in a series of blog posts beginning with “To Be or Not to Be (an Iterator),” Niebler, 2015 [3].) This document does not reflect the results of that research.

Whether and how to best support proxy iterators is left as future work.

C.2 Iterator Range Type

This paper does not define a concrete type for storing an iterator and a sentinel that models the Range concept. Such a type, like that presented in “A minimal std::range<Iter>,” [7] by J. Yasskin would be an obvious addition. Algorithms like equal_range and rotate could use a concrete range type instead of pair as their return type, improving composability. It would also be useful to implement a view::all range view that yields a lightweight range object that refers to all the elements of a container.

A future paper will propose such a type.

C.3 Range Views and Actions

The vast majority of the power of ranges comes from their composability. Views on existing Ranges can combine in chains of operations that evaluate lazily, giving programmers a concise and efficient way to express rich transformations on data. This paper is narrowly focused on the concepts and the algorithms, leaving range views as critically important future work.

If range views are composable, non-mutating, lazy algorithms over ranges, then range actions are composable, (potentially) mutating, eager algorithms over ranges. Such actions would allow users to send a container through a pipeline to sort it and remove the non-unique elements, for instance. This is something the range views cannot do. Range actions sit along side the views in the programmers toolbox.

C.4 Range Facade and Adaptor Utilities

Until it becomes trivial for users to create their own iterator types, the full potential of iterators will remain unrealized. The range abstraction makes that achievable. With the right library components, it should be possible for users to define a range with a minimal interface (e.g., current, done, and next members), and have iterator types automatically generated. Such a range facade class template is left as future work.
Another common need is to adapt an existing range. For instance, a lazy `transform` view should be as simple as writing an adaptor that customizes the behavior of a range by passing each element through a transformation function. The specification of a range `adaptor` class template is also left as future work.

### C.5 Infinite Ranges

It is not hard to define a type that represents an “infinite” range. Imagine a random number generator that keeps generating new random numbers for every application of `operator++` and `operator*` of its iterator. Indeed, the very existence of the `unreachable` sentinel type – which this document proposes – encourages users to think about some ranges as “infinite”. The sad reality is that infinite ranges are disallowed by the iterator concept hierarchy as defined in this document.

The problem with infinite ranges is the requirement that `WeaklyIncrementable` types have an associated `DifferenceType`. The `DifferenceType` must be a built-in integral type, and it must be large enough to hold the distance between any two valid iterators. This is implicit in the fact that the `distance` iterator primitive returns objects of type `DifferenceType`.

Given the fact that there are no infinite precision built-in integral types, the presence of `DifferenceType` in the concept requirements places a hard upper limit on how big ranges can be. This is a severe limitation.

Some observations:

1. Not all possible ranges have finitely countable elements.
2. Even a range with finitely countable elements may not be countable within the precision of the built-in integral types.
3. Not all algorithms require the ability to count increments. Algorithms like `distance` and `count` require finitely countable iterators, but `find` and `accumulate` do not.

The above observations suggest that there is a factorization of the existing concept hierarchy that could solve the problem. Some `Incrementables` are finitely countable with a built-in integral `DifferenceType`, and some are not. The algorithms that require countability must say so with an extra requirement.

Obviously, some algorithms like `accumulate` should never be asked to operate on a truly infinite sequence, even if they don’t actually maintain counters internally. Such algorithms could still be used with “possibly” infinite sequences. Note that the standard already defines such possibly infinite sequences; for instance, there is no limit in principle to the number of elements in a range delimited with `istream iterators`. It’s not hard to imagine other useful possibly-infinite range types: an infinite range that is delimited with a predicate or a sentinel value, for instance. The algorithm requirements should merely enforce that there is no integer overflow possible if the range happens to be larger than can be represented, not necessarily whether the algorithm will terminate or not.

### C.6 Common Type

The all-important `Common` concept relies on the existence of a SFINAE-friendly `common_type` trait, which did not make it into C++14. Solving the outstanding issues (active issues: #2460, #2465; and defects: #2408) with `common_type` is left as future work on which the correctness of this document depends.

### C.7 Numeric Algorithms and Containers

The numeric algorithms must also be constrained, and additional range-based overloads must be added. Also, the containers should be made range-aware; that is, they should have constructors and insert and append member functions that accept range arguments. These things can be done in a separate proposal.

### C.8 Verbosisty in Algorithm Constraints

The constraints of some of the algorithms are verbose. Some composite concepts exist that group constraints together to increase uniformity, aid comprehensibility, and reduce verbosity for those algorithms that use...
them. See `Sortable`, `Mergeable`, and `Permutable`, for instance. There may be other useful clusters of concepts for other algorithm groups. Finding them and using them to improve the algorithm constraints is an important work item.

### C.9 Initializer Lists

Algorithms that do not mutate their input should accept `initializer_list` arguments wherever `Iterables` are allowed. This requires extra overloads in places.
Annex D  (informative)
Reference implementation for tagged  [tagged]

Below is a reference implementation of the tagged class template described in 20.15.2, and also tagged_pair (20.15.5), tagged_tuple (20.15.6), and tag::in (25.1).

```cpp
namespace std { namespace experimental { namespace ranges_v1 { inline namespace v1 {

namespace tag { struct __specifier_tag { }; }

template <class T> struct __tag_spec { }; template <class Spec, class Arg> struct __tag_spec<Spec(Arg)> { using type = Spec; };

template <class T> struct __tag_elem { }; template <class Spec, class Arg> struct __tag_elem<Spec(Arg)> { using type = Arg; };

template <class T> concept bool TagSpecifier() {
    return DerivedFrom<T, tag::__specifier_tag>();
}

template <class T> concept bool TaggedType() {
    return requires {
        typename __tag_spec<T>::type;
        requires TagSpecifier<typename __tag_spec<T>::type>();
    };
}

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

template <class Base, class...Tags>
struct tuple_size<experimental::ranges_v1::tagged<Base, Tags...>> : tuple_size<Base> { }; template <size_t N, class Base, class...Tags>
struct tuple_element<N, experimental::ranges_v1::tagged<Base, Tags...>> : tuple_element<N, Base> { };

namespace experimental { namespace ranges_v1 { inline namespace v1 {

struct __getters { namespace tag { struct __specifier_tag { }; }

private:
    template <class, class TagSpecifier...>
    requires sizeof...(Tags) >= tuple_size<Base>::value
    friend struct tagged;
}}
```

Reference implementation for tagged
template <class Type, class Indices, class TagSpecifier...Tags>
struct collect_;

template <class Type, std::size_t...Is, class TagSpecifier...Tags>
struct collect_<Type, index_sequence<Is...>, Tags...>
: Tags::template getter<Type, Is...> {
    collect_() = default;
    collect_(const collect_&) = default;
    collect_& operator=(const collect_&) = default;
private:
    template <class Base, class TagSpecifier...Tags>
    requires sizeof...(Tags) <= tuple_size<Base>::value
    friend struct tagged;
    ~collect_() = default;
};

template <class Type, class TagSpecifier...Tags>
using collect = collect_<Type, make_index_sequence<sizeof...(Tags)>, Tags...>;

};

template <class Base, class TagSpecifier...Tags>
struct tagged
: Base, __getters::collect<tagged<Base, Tags...>, Tags...> {
    using Base::Base;
    tagged() = default;
    tagged(tagged&&) = default;
    tagged(tagged&) = default;
    tagged &operator=(tagged&) = default;
    tagged &operator=(const tagged&) = default;
    template <typename Other>
    requires Constructible<Base, Other>()
    tagged(tagged<Other, Tags...>&& that) :
    Base(static_cast<Other &&>(that)) {} 
    template <typename Other>
    requires Constructible<Base, const Other&>()
    tagged(const tagged<Other, Tags...>& that) :
    Base(static_cast<const Other&>(that)) {} 
    template <typename Other>
    requires Assignable<Base&, Other>()
    tagged& operator=(tagged<Other, Tags...>&& that) {
    static_cast<Base&>(*this) = static_cast<Other&&>(that);
    return *this;
    } 
    template <typename Other>
    requires Assignable<Base&, const Other&>()
    tagged& operator=(const tagged<Other, Tags...>& that) {
    static_cast<Base&>(*this) = static_cast<const Other&>(that);
    return *this;
    } 
    template <class U>
    requires Assignable<Base&, U>()
    tagged& operator=(U&& u) {
    static_cast<Base&>(*this) = std::forward<U>(u);
    return *this;
    }
};

Reference implementation for tagged

200
template <class TaggedType F, class TaggedType S>
using tagged_pair =
tagged<pair<typename __tag_elem<F>::type, typename __tag_elem<S>::type>,
    typename __tag_spec<F>::type, typename __tag_spec<S>::type>;

template <class TaggedType...Types>
using tagged_tuple =
tagged<tuple<typename __tag_elem<Types>::type...>,
    typename __tag_spec<Types>::type...>;

namespace tag {
    struct in : __specifier_tag {
    private:
        friend struct __getters;
        template <class Derived, size_t I>
        struct getter {
            getter() = default;
            getter(const getter&) = default;
            getter &operator=(const getter&) = default;
            constexpr decltype(auto) in() & {
                return get<I>(static_cast<Derived &>(*this));
            }
            constexpr decltype(auto) in() && {
                return get<I>(static_cast<Derived &&>(*this));
            }
            constexpr decltype(auto) in() const & {
                return get<I>(static_cast<const Derived &>(*this));
            }
        private:
            friend struct __getters;
            ~getter() = default;
        };
    } // Other tag specifiers defined similarly, see 25.1
};
}
Bibliography


Index

x C++ Standard, 2
“The Palo Alto”, 2

Concepts TS, 2
constant iterator, 34

multi-pass guarantee, 41
mutable iterator, 34

requirements, 6
   iterator, 33
      uniform random number generator, 189

statement
   iteration, 5
swappable, 12
swappable with, 12

undefined behavior, 101
uniform random number generator
   requirements, 189
unspecified, 167

Reference implementation for tagged
Index of library names

adjacent_find, 143
advance, 62, 93
all_of, 139
any_of, 140
Assignable, 11
back_insert_iterator, 70
    back_insert_iterator, 70
    constructor, 71
back_inserter, 71
base
    counted_iterator, 89
    move_iterator, 77
    reverse_iterator, 67
begin(C&), 110
begin(T (&)[N]), 110
bidirectional_iterator_tag, 61
BidirectionalIterator, 41
binary_search, 171
Boolean, 13

cbegin(const C&), 110
cend(const C&), 110
Common, 10
common_iterator, 80
    common_iterator, 82
    constructor, 82
dangling, 93
dangling, 94
destructor, 83
operator!=, 84
operator*, 83
operator++, 84
operator--, 84
operator->, 83
operator=, 83
operator==, 84
count, 96
counted_iterator, 86
crbegin(const C& c), 111
crend(const C& c), 111
dangling, 93
dangling, 94
destructor, 83
empty, 61

Reference implementation for tagged

© ISO/IEC
N4560
Reference implementation for tagged
mismatch, 144
Movable, 18
move, 151
move_backward, 151
move_iterator, 74
    base, 77
    constructor, 76
    move_iterator, 76
    operator!=, 78
    operator*, 77
    operator+, 78, 79
    operator++, 77
    operator+=, 78
    operator-, 78, 79
    operator-=, 78
    operator->, 77
    operator==, 77
    operator==, 78
    operator>, 79
    operator>=, 78
    operator[]', 79
    operator[[]', 78
MoveConstructible, 17
MoveMergeable, 45
MoveWritable, 35

next, 63
next_permutation, 186
none_of, 140
not_equal_to, 25
not_equal_to<>, 26
nth_element, 169

operator!=
    common_iterator, 84
    counted_iterator, 91
default_sentinel, 85
istream_iterator, 99
istreambuf_iterator, 103
move_iterator, 78
reverse_iterator, 68
unreachable, 95
operator*
    back_insert_iterator, 71
    common_iterator, 83
counted_iterator, 89
front_insert_iterator, 72
insert_iterator, 74
istream_iterator, 98, 99
istreambuf_iterator, 103
move_iterator, 77
ostream_iterator, 100
ostreambuf_iterator, 104
reverse_iterator, 67
operator+
    counted_iterator, 90, 93
    move_iterator, 78, 79
    reverse_iterator, 67, 69
operator++
    back_insert_iterator, 71
    common_iterator, 84
counted_iterator, 89
front_insert_iterator, 72
insert_iterator, 74
istream_iterator, 98, 99
istreambuf_iterator, 103
move_iterator, 77
ostream_iterator, 100
ostreambuf_iterator, 104
reverse_iterator, 67
operator++=
    counted_iterator, 90
    move_iterator, 78
    reverse_iterator, 68
operator-
    common_iterator, 84
counted_iterator, 90, 92
default_sentinel, 85
move_iterator, 78, 79
reverse_iterator, 68, 69
operator-=
    counted_iterator, 90
    move_iterator, 78
    reverse_iterator, 68
operator->
    common_iterator, 83
    istream_iterator, 98
    move_iterator, 77
    reverse_iterator, 67
operator--
    counted_iterator, 89
    move_iterator, 77
    reverse_iterator, 68
operator<
    counted_iterator, 91
default_sentinel, 85
    move_iterator, 78
    reverse_iterator, 68
unreachable, 95
operator<=
counted_iterator, 92
default_sentinel, 85
move_iterator, 78
reverse_iterator, 69
unreachable, 96
operator=
  reverse_iterator, 66
back_insert_iterator, 71
common_iterator, 83
counted_iterator, 89
front_insert_iterator, 72
insert_iterator, 73
move_iterator, 77
ostream_iterator, 100
ostreambuf_iterator, 104
tagged, 29
operator==
  common_iterator, 84
counted_iterator, 91
default_sentinel, 85
istream_iterator, 99
istreambuf_iterator, 103
move_iterator, 78
reverse_iterator, 68
unreachable, 95
operator>
  counted_iterator, 92
default_sentinel, 85
move_iterator, 79
reverse_iterator, 69
unreachable, 96
operator>=
  counted_iterator, 92
default_sentinel, 85
move_iterator, 79
reverse_iterator, 69
unreachable, 96
operator[]
  counted_iterator, 90
move_iterator, 78
reverse_iterator, 68
ostream_iterator, 99
  constructor, 100
destructor, 100
operator*, 100
operator++, 100
operator=, 100
ostreambuf_iterator, 103
  constructor, 104
output_iterator_tag, 61
OutputIterator, 40
partial_sort, 167
partial_sort_copy, 167
partition, 162
partition_copy, 163
partition_move, 164
partition_point, 164
Permutable, 45
pop_heap, 179
Predicate, 19
prev, 64
prev_permutation, 186
projected, 44
proxy
  istreambuf_iterator, 102
push_heap, 178
random_access_iterator_tag, 61
RandomAccessIterator, 42
rbegin(C&), 110
rbegin(initializer_list<E>), 111
rbegin(T (&array)[N]), 111
Readable, 34
Regular, 19
RegularFunction, 19
Relation, 20
remove, 157
remove_copy, 158
remove_copy_if, 158
remove_if, 157
rend(const C&), 110
rend(initializer_list<E>), 111
rend(T (&array)[N]), 111
replace, 154
replace_copy, 155
replace_copy_if, 155
replace_if, 154
reverse, 160
reverse_copy, 160
reverse_iterator, 64
  reverse_iterator, 66
base, 67
  constructor, 66
make_reverse_iterator non-member function, 69
  operator++, 67
  operator--, 67
rotate, 161
rotate_copy, 161
Same, 10
search, 148
Reference implementation for tagged
  207
search_n, 148
Semiregular, 18
Sentinel, 39
set_difference, 176
set_intersection, 176
set_symmetric_difference, 177
set_union, 175
shuffle, 161
SignedIntegral, 11
size, 113
SizedIteratorRange, 46
sort, 166
sort_heap, 180
Sortable, 46
stable_partition, 163
stable_sort, 166
StrictTotallyOrdered, 15
swap, 22
    tagged, 30
    tagged, 30
swap_ranges, 152
Swappable, 11

tagged, 27
    operator=, 29
    swap, 30
tagged, 29
tagged_tuple
    make_tagged_tuple, 32
transform, 152
tuple_element, 30
tuple_size, 30

unique, 159
unique_copy, 159
unreachable, 94
    operator!=, 95
    operator<, 95
    operator<=, 96
    operator==, 95
    operator>, 96
    operator>={}, 96
UnsignedIntegral, 11
upper_bound, 170

value_type_t, 35

weak_input_iterator_tag, 61
WeakInputIterator, 39
WeakIterator, 38
WeaklyIncrementable, 37

Reference implementation for tagged