Document Number: P0459R0
Date: 2016-10-15
Reply to: Eric Niebler
eric.niebler@gmail.com
Reply to: Casey Carter
casey@carter.net

C++ Extensions for Ranges, Speculative Combined Proposal Document[1]

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

1) This document combines the text of the current Ranges TS working draft N4560 with the wording of proposals P0370 and P0441 for review purposes.

Contents

List of Tables

List of Figures

1 General [intro]

> 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 [intro.scope]

1 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:

(1.1) — The reformulation of the foundational and iterator concept requirements using the syntax of the Concepts TS (1.2).

(1.2) — The respecification of the Standard Library algorithms in terms of the new concepts.

(1.3) — 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.

(1.4) — The addition of new concepts describing *range* and *view* abstractions; that is, objects with a begin iterator and an end sentinel.

(1.5) — The addition of new overloads of the Standard Library algorithms that take iterable objects.

(1.6) — Support of *callable objects* (as opposed to *function objects*) passed as arguments to the algorithms.

(1.7) — The addition of optional *projection* arguments to the algorithms to permit on-the-fly data transformations.

(1.8) — Changes to existing iterator primitives and new primitives in support of the addition of sentinels to the library.

(1.9) — Changes to the existing iterator adaptors and stream iterators to make them model the new iterator concepts.

(1.10) — New iterator adaptors (`counted_iterator` and `common_iterator`) and sentinels (`unreachable`).

2 Changes to the core language include:

(2.1) — the extension of the range-based `for` statement to support the new iterator range requirements (24.10).

3 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).

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

1.2 References [intro.refs]

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

(1.1) — ISO/IEC 14882:2014, *Programming Languages - C++*

(1.2) — ISO/IEC TS 19217:2015, *Programming Languages - C++ Extensions for Concepts*

(1.3) — JTC1/SC22/WG21 N4128, *Ranges for the Standard Library, Revision 1*

(1.4) — 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 ISO/IEC TS 19217:2105 is called the *Concepts TS*.

1.3 Implementation compliance [intro.compliance]

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

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

1 Since the extensions described in this 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:

(1.1) — modify an existing interface in the C++ Standard Library in-place,

(1.2) — are declared in namespace `std::experimental::ranges::v1`.

[Editor's note: The following text 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 ISO/IEC TS 19217:2015 (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 previous working draft of this document, and ~~strikethrough~~ denotes text removed.

4 This paper assumes that the contents of the `std::experimental::ranges::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.

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

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

```
#include <algorithm>
```

Table 1 — Ranges TS library headers

`<experimental/ranges/algorithm>`	`<experimental/ranges/random>`
`<experimental/ranges/concepts>`	`<experimental/ranges/tuple>`
`<experimental/ranges/functional>`	`<experimental/ranges/utility>`
`<experimental/ranges/iterator>`	

6 Statements [stmt]

6.5 Iteration statements [stmt.iter]

6.5.4 The range-based `for` statement [stmt.ranged]

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

1 For a range-based for statement of the form

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

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

`(` *expression* `)`

and for a range-based for statement of the form

`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

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

```
{
  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:

(1.1) — 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;

(1.2) — if `_RangeT` is a class type, the *unqualified-id*s `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;

(1.3) — 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:*

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

— *end example*]

17 Library introduction [library]

17.5.1.3 Requirements [structure.requirements]

1 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

2 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.

3 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.

4 Requirements are stated in terms of concepts (Concepts TS [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 formal type parameters.

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

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

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

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

9 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 [customization.point.object]

1 A *customization point object* is a function object (20.9) with a literal class type that interacts with user-defined types while enforcing semantic requirements on that interaction.

2 The type of a customization point object shall satisfy `Semiregular` (19.4.8).

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

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

4 The type of a customization point object `T` shall satisfy `Callable<const T, Args...>()` (19.5.2) when the types of `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.

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

6 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.

7 [*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 `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 `std`. When the deleted overloads are viable, user-defined overloads must be more specialized (14.5.6.2) or more constrained (Concepts TS [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 subsequent chapters should be renumbered as appropriate, but they aren't here for the sake of simplicity).]

19.1 General [concepts.lib.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

	Subclause	Header(s)
19.2	Core language concepts	`<experimental/ranges/concepts>`
19.3	Comparison concepts	
19.4	Object concepts	
19.5	Callable concepts	

3 The concepts in this Clause are defined in the namespace `std::experimental::ranges::v1`.

19.1.1 Equality Preservation [concepts.lib.general.equality]

1 An expression is *equality preserving* if, given equal inputs, the expression results in equal outputs. The inputs to an expression are the set of the expression's operands. The output of an expression is the expression's result and all operands modified by the expression.

2 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. The *domain* of an expression is the set of input values for which the expression is required to be well-defined.

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.

5 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.

6 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

```
a == d;       a == b;             a == move(b);       a == d;
c == a;       c == move(a);       c == move(d);
move(a) == d; move(a) == b;       move(a) == move(b); move(a) == move(d);
move(c) == b; move(c) == move(b); move(c) == d;       move(c) == move(d);
```

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:

```
struct T {
  bool operator==(const T&) const { return true; }
  bool operator==(T&) = delete;
};
```

`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*]

19.2 Core language concepts [concepts.lib.corelang]

19.2.1 In general [concepts.lib.corelang.general]

1 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 [concepts.lib.corelang.same]

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

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

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

19.2.3 Concept DerivedFrom [concepts.lib.corelang.derived]

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

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

19.2.4 Concept ConvertibleTo [concepts.lib.corelang.convertibleto]

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

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

19.2.5 Concept Common [concepts.lib.corelang.common]

1 If `T` and `U` can both be explicitly converted to some third type, `C`, then `T` and `U` share a *common type*, `C`. [*Note:* `C` could be the same as `T`, or `U`, or it could be a different type. `C` may not be unique. — *end note*]

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

2 Let `C` be `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

(2.1) — `C(t1)` equals `C(t2)` if and only if `t1` equals `t2`.

(2.2) — `C(u1)` equals `C(u2)` if and only if `u1` equals `u2`.

3 [*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;
}
```

1 [*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>();
}
```

1 [*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>
concept bool Assignable() {
  return Common<T, U>() && requires(T&& t, U&& u) {
    { std::forward<T>(t) = std::forward<U>(u) } -> Same<T&>;
  };
}
```

1 Let `t` be an lvalue of type `T`, and `R` be the type `remove_reference_t<U>`. If `U` is an lvalue reference type, let `v` be an lvalue of type `R`; otherwise, let `v` be an rvalue of type `R`. Let `uu` be a distinct object of type `R` such that `uu == v`. Then `Assignable<T, U>()` is satisfied if and only if

(1.1) — `std::addressof(t = v) == std::addressof(t)`.

(1.2) — After evaluating `t = v`:

(1.2.1) — `t == uu`.

(1.2.2) — If `v` is a non-`const` rvalue, its resulting 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. — *end note*]

(1.2.3) — Otherwise, `v` is not modified.

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) {
    ranges::swap(std::forward<T>(a), std::forward<T>(b));
  };
}

template <class T, class U>
concept bool Swappable() {
  return Swappable<T>() &&
    Swappable<U>() &&
```

```
    Common<T, U>() &&
    requires(T&& t, U&& u) {
      ranges::swap(std::forward<T>(t), std::forward<U>(u));
      ranges::swap(std::forward<U>(u), std::forward<T>(t));
    };
}
```

1 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`.

2 An object `t` is *swappable with* an object `u` if and only if `Swappable<T, U>()` is satisfied. `Swappable<T, U>()` is satisfied if and only if given distinct objects `tt` equal to `t` and `uu` equal to `u`, after evaluating either `ranges::swap(t, u)` or `ranges::swap(u, t)`, `tt` is equal to `u` and `uu` is equal to `t`.

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

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

```
#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]

1 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]

1 The `Boolean` concept specifies 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;
      { a != b1 } -> bool;
    };
}
```

2 Given values `b1` and `b2` of type `B`, then `Boolean<B>()` is satisfied if and only if

(2.1) — `bool(b1) == [](bool x) { return x; }(b1)`.

(2.2) — `bool(b1) == !bool(!b1)`.

(2.3) — `(b1 && b2)`, `(b1 && bool(b2))`, and `(bool(b1) && b2)` are all equal to `(bool(b1) && bool(b2))`, and have the same short-circuit evaluation.

(2.4) — `(b1 || b2)`, `(b1 || bool(b2))`, and `(bool(b1) || b2)` are all equal to `(bool(b1) || bool(b2))`, and have the same short-circuit evaluation.

(2.5) — `bool(b1 == b2)`, `bool(b1 == bool(b2))`, and `bool(bool(b1) == b2)` are all equal to `(bool(b1) == bool(b2))`.

(2.6) — `bool(b1 != b2)`, `bool(b1 != bool(b2))`, and `bool(bool(b1) != b2)` are all equal to `(bool(b1) != bool(b2))`.

3 [*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].]

```
template <class T, class U>
concept bool WeaklyEqualityComparable() {
  return requires(const T& t, const U& u) {
    { t == u } -> Boolean;
    { u == t } -> Boolean;
    { t != u } -> Boolean;
    { u != t } -> Boolean;
  };
}
```

1 Let `t` and `u` be objects of types `T` and `U`. `WeaklyEqualityComparable<T, U>()` is satisfied if and only if:

(1.1) — `t == u`, `u == t`, `t != u`, and `u != t` have the same domain.

(1.2) — `bool(u == t) == bool(t == u)`.

(1.3) — `bool(t != u) == !bool(t == u)`.

(1.4) — `bool(u != t) == bool(t != u)`.

```
template <class T>
concept bool EqualityComparable() {
  return WeaklyEqualityComparable<T, T>();
}
```

2 Let `a` and `b` be objects of type `T`. `EqualityComparable<T>()` is satisfied if and only if:

(2.1) — `bool(a == b)` if and only if `a` is equal to `b`.

3 [*Note:* The requirement that the expression `a == b` is equality preserving implies that `==` is reflexive, transitive, and symmetric. — *end note*]

```
template <class T, class U>
concept bool EqualityComparable() {
  return Common<T, U>() &&
    EqualityComparable<T>() &&
    EqualityComparable<U>() &&
    EqualityComparable<common_type_t<T, U>>() &&
    WeaklyEqualityComparable<T, U>();
}
```

4 Let `a` be an object of type `T`, `b` be an object of type `U`, and `C` be `common_type_t<T, U>`. Then `EqualityComparable<T, U>()` is satisfied if and only if:

(4.1) — `bool(a == b) == bool(C(a) == C(b))`.

5 [*Note:* The distinction between `EqualityComparable<T, U>()` and `WeaklyEqualityComparable<T, U>()` is purely semantic. — *end note*]

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]]

```
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;
    };
}
```

1 Let `a`, `b`, and `c` be objects of type `T`. Then `StrictTotallyOrdered<T>()` is satisfied if and only if

(1.1) — Exactly one of `bool(a < b)`, `bool(b < a)`, or `bool(a == b)` is `true`.

(1.2) — If `bool(a < b)` and `bool(b < c)`, then `bool(a < c)`.

(1.3) — `bool(a > b) == bool(b < a)`.

(1.4) — `bool(a <= b) == !bool(b < a)`.

(1.5) — `bool(a >= b) == !bool(a < b)`.

```
template <class T, class U>
concept bool StrictTotallyOrdered() {
  return Common<T, U>() &&
    StrictTotallyOrdered<T>() &&
    StrictTotallyOrdered<U>() &&
    StrictTotallyOrdered<common_type_t<T, U>>() &&
    EqualityComparable<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;
    };
}
```

2 Let `t` be an object of type `T`, `u` be an object of type `U`, and `C` be `common_type_t<T, U>`. Then `StrictTotallyOrdered<T, U>()` is satisfied if and only if

(2.1) — `bool(t < u) == bool(C(t) < C(u))`.

(2.2) — `bool(t > u) == bool(C(t) > C(u))`.

(2.3) — `bool(t <= u) == bool(C(t) <= C(u))`.

(2.4) — `bool(t >= u) == bool(C(t) >= C(u))`.

(2.5) — `bool(u < t) == bool(C(u) < C(t))`.

(2.6) — `bool(u > t) == bool(C(u) > C(t))`.

(2.7) — `bool(u <= t) == bool(C(u) <= C(t))`.

(2.8) — `bool(u >= t) == bool(C(u) >= C(t))`.

19.4 Object concepts [concepts.lib.object]

1 This section describes concepts that specify 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 specify features of types to describe families of object types.]

19.4.1 Concept Destructible [concepts.lib.object.destructible]

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

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

```
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;
  };
}
```

2 The expression requirement `&ct` does not require implicit expression variants.

3 Given a (possibly `const`) lvalue `t` of type `T` and pointer `p` of type `T*`, `Destructible<T>()` is satisfied if and only if

(3.1) — After evaluating the expression `t.~T()`, `delete p`, or `delete[] p`, all resources owned by the denoted object(s) are reclaimed.

(3.2) — `&t == std::addressof(t)`.

(3.3) — The expression `&t` is non-modifying.

19.4.2 Concept Constructible [concepts.lib.object.constructible]

1 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.

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

1 [*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>&&>() &&
    ConvertibleTo<remove_cv_t<T>&&, T>();
}
```

1 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>&>() &&
    ConvertibleTo<remove_cv_t<T>&, T>() &&
    ConvertibleTo<const remove_cv_t<T>&, T>() &&
    ConvertibleTo<const remove_cv_t<T>&&, T>();
}
```

1 Let `v` be an lvalue of type (possibly `const`) `remove_cv_t<T>` or an rvalue of type `const remove_cv_t<T>`. Then `CopyConstructible<T>()` is satisfied if and only if

(1.1) — After the definition `T u = v;`, `v` is equal to `u`.

(1.2) — `T{v}` or `*new T{v}` is equal to `v`.

[Editor's note: Ideally, `CopyConstructible` would include an array allocation requirement `new T[1]{t}`. This is not currently possible since [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 [moveassignable] with references to [concepts.lib.object.movable].]

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

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

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

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

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

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

1 [*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>();
}
```

1 [*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 Callable concepts [concepts.lib.callables]

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

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

19.5.2 Concept Callable [concepts.lib.callables.callable]

1 The `Callable` concept specifies a relationship between a callable type (20.9.1) `F` and a set of argument types `Args...` which can be evaluated by the library function `invoke` (20.9.3).

```
template <class F, class... Args>
concept bool Callable() {
  return CopyConstructible<F>() &&
```

```
    requires(F f, Args&&... args) {
      invoke(f, std::forward<Args>(args)...); // not required to be equality preserving
    };
}
```

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

19.5.3 Concept RegularCallable [concepts.lib.callables.regularcallable]

```
template <class F, class... Args>
concept bool RegularCallable() {
  return Callable<F, Args...>();
}
```

1 The `invoke` function call expression shall be equality-preserving (19.1.1). [*Note:* This requirement supersedes the annotation in the definition of `Callable`. — *end note*]

2 [*Note:* A random number generator does not satisfy `RegularCallable`. — *end note*]

3 [*Note:* The distinction between `Callable` and `RegularCallable` is purely semantic. — *end note*]

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

```
template <class F, class... Args>
concept bool Predicate() {
  return RegularCallable<F, Args...>() &&
    Boolean<result_of_t<F&(Args...)>>();
}
```

19.5.5 Concept Relation [concepts.lib.callables.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>() &&
    Relation<R, U>() &&
    Common<T, U>() &&
    Relation<R, common_type_t<T, U>>() &&
    Predicate<R, T, U>() &&
    Predicate<R, U, T>();
}
```

1 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 `common_type_t<T, U>`. Then `Relation<R, T, U>()` is satisfied if and only if

(1.1) — `bool(r(a, b)) == bool(r(C(a), C(b)))`.

(1.2) — `bool(r(b, a)) == bool(r(C(b), C(a)))`.

19.5.6 Concept StrictWeakOrder [concepts.lib.callables.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>();
}
```

1 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].]

2 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:

(2.1) — `comp(a, b) && comp(b, c)` implies `comp(a, c)`

(2.2) — `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)` [*Note:* Under these conditions, it can be shown that

(2.2.1) — `equiv` is an equivalence relation

(2.2.2) — `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

(2.2.3) — The induced relation is a strict total ordering. — *end note*]

20 General utilities library [utilities]

20.2 Utility components [utility]

1 **Header <experimental/ranges/utility> synopsis**

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 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)));
  namespace {
    constexpr unspecified swap = unspecified;
  }

  // 20.2.3, exchange:
  template <classMoveConstructible 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;

  // 20.15.4, tagged pairs
  template <TaggedType T1, TaggedType T2> using tagged_pair = see below;

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

namespace std {
  // 20.15.3, tuple-like access to tagged
  template <class Base, class... Tags>
  struct tuple_size<experimental::ranges::tagged<Base, Tags...>>;

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

2 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/utility>` are imported with *using-declarations* (7.3.3). [*Example:*

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  using std::pair;
  using std::make_pair;
  // ... others
}}}}
```

— *end example*]

20.2.2 swap [utility.swap]

1 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 `MoveConstructible<T>()` and `Assignable<T&, 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 either `MoveConstructible<T>()` or `Assignable<T&, 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.

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

```
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 <classMoveConstructible 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;
```

20.9 Function objects [function.objects]

2 **Header <experimental/ranges/functional> synopsis**

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 20.9.3, invoke:
  template <class F, class... Args>
  result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

  // 20.9.5, comparisons:
  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;

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

  // 20.9.13, identity:
  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` in header `<experimental/ranges/functional>` are imported with *using-declarations* (7.3.3). [*Example:*

```
namespace std { namespace experimental { namespace ranges { inline namespace 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` in header `<experimental/ranges/functional>` are aliased with a *namespace-alias-definition* (7.3.2). [*Example:*

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

— *end example*]

[Editor's note: Before [refwrap], insert the following section and renumber subsequent sections as appropriate. (Renumbering hasn't been performed herein to ease review.)]

20.9.3 Function template invoke [func.invoke]

```
template <class F, class... Args>
result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);
```

1 *Effects:* Equivalent to *`INVOKE`*`(std::forward<F>(f), std::forward<Args>(args)...)` (20.9.2).

20.9.5 Comparisons [comparisons]

1 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;
};
```

2 `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;~~
};
```

3 `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;~~
};
```

4 `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;~~
};
```

5 `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;~~
};
```

6 `operator()` returns `x >= y`.

```
template <class T = void>
  requires StrictTotallyOrdered<T>() || Same<T, void>()
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;~~
};
```

7 `operator()` returns `x <= 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;
};
```

8 `operator()` returns `std::forward<T>(t) == std::forward<U>(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;
};
```

9 `operator()` returns `std::forward<T>(t) != std::forward<U>(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;
};
```

10 `operator()` returns `std::forward<T>(t) > std::forward<U>(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;
};
```

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

```
template <> struct greater_equal<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;
};
```

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

```
template <> struct less_equal<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;
};
```

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;
};
```

1 `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]

1 The library provides a template for augmenting a tuple-like type with named element accessor member functions. The library also provides several templates that provide access to `tagged` objects as if they were `tuple` objects (see 20.4.2.6).

[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]

1 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` (20.15.4) and `tagged_tuple` (20.15.5).

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

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

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class T>
  concept bool TagSpecifier() {
    return implementation-defined;
  }

  template <class F>
  concept bool TaggedType() {
    return implementation-defined;
```

```
  }

  template <class Base, TagSpecifier... Tags>
    requires sizeof...(Tags) <= tuple_size<Base>::value
  struct tagged :
    Base, TAGGET(tagged<Base, Tags...>, Ti, i)... { // see below
    using Base::Base;
    tagged() = default;
    tagged(tagged&&) = default;
    tagged(const tagged&) = default;
    tagged &operator=(tagged&&) = default;
    tagged &operator=(const tagged&) = default;
    template <class Other>
      requires Constructible<Base, Other>()
    tagged(tagged<Other, Tags...> &&that) noexcept(see below);
    template <class Other>
      requires Constructible<Base, const Other&>()
    tagged(const tagged<Other, Tags...> &that);
    template <class Other>
      requires Assignable<Base&, Other>()
    tagged& operator=(tagged<Other, Tags...>&& that) noexcept(see below);
    template <class Other>
      requires Assignable<Base&, const Other&>()
    tagged& operator=(const tagged<Other, Tags...>& that);
    template <class U>
      requires Assignable<Base&, U>() && !Same<decay_t<U>, tagged>()
    tagged& operator=(U&& u) noexcept(see below);
    void swap(tagged& that) noexcept(see below)
      requires Swappable<Base&>();
    friend void swap(tagged&, tagged&) noexcept(see below)
      requires Swappable<Base&>();
  };
}}}}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14 *Returns:* `*this`.

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

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

16 *Returns:* `*this`.

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

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

```
is_nothrow_assignable<Base&, U>::value
```

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

19 *Returns:* `*this`.

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

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

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

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

22 *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&>();
```

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

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

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

25 *Throws:* Nothing unless the call to `lhs.swap(rhs)` throws.

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

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

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

20.15.4 Alias template tagged_pair [tagged.pairs]

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

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

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*]

20.15.4.1 Tagged pair creation functions [tagged.pairs.creation]

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

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

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

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

3 [*Example:* In place of:

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

a C++ program may contain:

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

— end example]

20.15.5 Alias template tagged_tuple [tagged.tuple]

Header <experimental/ranges/tuple> synopsis

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

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

1 Any entities declared or defined in namespace `std` in header `<tuple>` that are not already defined in namespace `std::experimental::ranges` in header `<experimental/ranges/tuple>` are imported with *using-declarations* (7.3.3). [*Example:*

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  using std::tuple;
  using std::make_tuple;
  // ... others
}}}}
```

— end example]

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

2 [*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));
```

— end example]

20.15.5.1 Tagged tuple creation functions [tagged.tuple.creation]

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

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

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

3 [*Example:*

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

creates a tagged tuple of type

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

— *end example*]

24 Iterators library [iterators]

24.1 General [iterators.general]

1 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 (24.9).

2 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

	Subclause	Header(s)
24.2	Requirements	
24.6	Iterator primitives	<experimental/ranges/iterator>
24.7	Predefined iterators	
24.8	Stream iterators	
24.9	Ranges	

24.2 Iterator requirements [iterator.requirements]

24.2.1 In general [iterator.requirements.general]

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

2 Since iterators are an abstraction of pointers, their semantics ~~is~~are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This International Standard defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 4.

Table 4 — Relations among iterator categories

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

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

4 Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used when-

ever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

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

6 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [*Example:* After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a singular value of a pointer. — *end example*] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value, the assignment of a non-singular value to an iterator that holds a singular value, and~~, 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.

7 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a *sentinel* that designate the beginning and end of the computation. An iterator and a sentinel denoting a range are comparable. A sentinel denotes an element when it compares equal to an iterator `i`, and `i` points to that element. The types of a sentinel and an iterator that denote a range must satisfy `Sentinel` (24.2.7).

8 ~~An iterator~~ A sentinel ~~`j`~~`s` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == `~~`j`~~`s`. If ~~`j`~~`s` is reachable from `i`, they ~~refer to elements of the same sequence~~denote a range.

9 ~~Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation.~~ A range ~~`[i,i)`~~ `[i,s)` is ~~an~~ empty ~~range~~ if `i == s`; ~~in general, a range `[i,j)`~~ otherwise, `[i,s)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the first iterator `j` such that `j == s`.

10 ~~R~~A range ~~`[i,j)`~~ `[i,s)` is valid if and only if ~~`j`~~`s` is reachable from `i`. The result of the application of functions in the library to invalid ranges is undefined.

11 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.~~

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

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

14 ~~In the following sections, `a` and `b` denote values of type `X` or `const X`, `difference_type` and `reference` refer to the types `iterator_traits<X>::difference_type` and `iterator_traits<X>::reference`, respectively, `n` denotes a value of `difference_type`, `u`, `tmp`, and `m` denote identifiers, `r` denotes a value of `X&`, `t` denotes a value of value type `T`, `o` denotes a value of some type that is writable to the output iterator. [*Note:* For an iterator type `X` there must be an instantiation of `iterator_traits<X>` (24.4.1). — *end note*]~~

24.2.2 Concept Readable [iterators.readable]

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

3) 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 value_type_t<I>;
      { *i } -> const value_type_t<I>&; // pre: i is dereferenceable
    };
}
```

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

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

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

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

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

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

3 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`.

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

5 When instantiated with a type `I` such that `I::value_type` is valid and denotes a type, `value_type<I>::type` names that type, unless it is not an object type (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*]

6 When instantiated with a type `I` such that `I::element_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:* 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 Writable [iterators.writable]

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

```
template <class Out, class T>
concept bool Writable() {
```

```
    return Semiregular<Out>() &&
      requires(Out o, T&& t) {
        *o = std::forward<T>(t); // not required to be equality preserving
      };
  }
```

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

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

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

4 If `E` is an xvalue (3.10), the resulting state of the object it denotes is unspecified. [*Note:* The object must still meet the requirements of any library component that is using it. The operations listed in those requirements must work as specified whether the object has been moved from or not. — *end note*]

[Editor's note: "If `E` is an xvalue" suffices for now, but incorporating P0022's proxy iterators will require a characterization of the "xvalue-ness" of a proxy reference type.]

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

24.2.4 Concept WeaklyIncrementable [iterators.weaklyincrementable]

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

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

2 Let `i` be an object of type `I`. When both pre- and post-increment are valid, `i` is said to be *incrementable*. Then `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++` advance `i` to the next element.

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

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

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

24.2.5 Concept Incrementable [iterators.incrementable]

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

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

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

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

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

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

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

[Editor's note: Section "Iterator" renamed to "Concept Iterator" below:]

24.2.6 Concept Iterator [iterators.iterator]

1 The `Iterator` ~~requirements~~concept forms the basis of the iterator concept taxonomy; every iterator satisfies the `Iterator` requirements. This ~~set of requirements~~concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (24.2.7), to read (24.2.9) or write (24.2.10) values, or to provide a richer set of iterator movements (24.2.11, 24.2.12, 24.2.13).)

[Editor's note: Remove para 2 and Table 106.]

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

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

24.2.7 Concept Sentinel [iterators.sentinel]

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

```
template <class S, class I>
concept bool Sentinel() {
  return Semiregular<S>() &&
    Iterator<I>() &&
```

```
    WeaklyEqualityComparable<S, I>();
}
```

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

(2.1) — `i == s` is well-defined.

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

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

24.2.8 Concept SizedSentinel [iterators.sizedsentinel]

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

```
template <class S, class I>
constexpr bool disable_sized_sentinel = false;

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

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

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

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

3 [*Note:* `disable_sized_sentinel` provides a mechanism to enable use of sentinels and iterators with the library that meet the syntactic requirements but do not in fact satisfy `SizedSentinel`. A program that instantiates a library template that requires `SizedSentinel` with an iterator type `I` and sentinel type `S` that meet the syntactic requirements of `SizedSentinel<S, I>()` but do not satisfy `SizedSentinel` is ill-formed with no diagnostic required unless `disable_sized_sentinel<S, I>` evaluates to `true` (17.5.1.3). — *end note*]

4 [*Note:* The `SizedSentinel` concept is satisfied by pairs of `RandomAccessIterator`s (24.2.13) and by counted iterators and their sentinels (24.7.6.1). — *end note*]

[Editor's note: Section "Input iterators" renamed to "Concept InputIterator" below:]

24.2.9 Concept InputIterator [iterators.input]

[Editor's note: Replace the entire content of the section with:]

1 The `InputIterator` concept is a refinement of `Iterator` (24.2.6). It defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (24.2.2)) and which can be both pre- and post-incremented. [*Note:* Unlike in ISO/IEC 14882, input iterators are not required to satisfy `EqualityComparable` (19.3.3). — *end note*]

```
template <class I>
concept bool InputIterator() {
  return Iterator<I>() &&
    Readable<I>() &&
    requires(I i, const I ci) {
      typename iterator_category_t<I>;
      requires DerivedFrom<iterator_category_t<I>, input_iterator_tag>();
      { i++ } -> Readable; // not required to be equality preserving
      requires Same<value_type_t<I>, value_type_t<decltype(i++)>>();
      { *ci } -> const value_type_t<I>&;
    };
}
```

[Editor's note: Section "Output iterators" renamed to "Concept OutputIterator" below:]

24.2.10 Concept OutputIterator [iterators.output]

[Editor's note: Remove para 1 and Table 108]

1 The `OutputIterator` concept is a refinement of `Iterator` (24.2.6). It defines requirements for a type that can be used to write values (from the requirement for `Writable` (24.2.3)) and which can be both pre- and post-incremented. However, output iterators are not required to satisfy `EqualityComparable`.

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

2 [*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 output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — *end note*]

24.2.11 Concept ForwardIterator [iterators.forward]

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

(1.1) — `X` satisfies the requirements of an input iterator (24.2.9),

(1.2) — X satisfies the `DefaultConstructible` requirements (17.6.3.1),

(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`,

(1.4) — the expressions in Table 109 are valid and have the indicated semantics, and

(1.5) — objects of type `X` offer the multi-pass guarantee, described below.

2 The `ForwardIterator` concept refines `InputIterator` (24.2.9), adding equality comparison and the multi-pass guarantee, specified below.

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

3 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:* ~~v~~Value-initialized iterators behave as if they refer past the end of the same empty sequence. — *end note*]

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

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

(4.2) — ~~`X` is a pointer type or t~~The expression ~~`(void)++X(a)`~~`([](X x){++x;}(a), *a)` is equivalent to the expression `*a`.

5 [*Note:* The requirement that `a == b` implies `++a == ++b` (which is not true for ~~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) allow~~s~~ the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

[Editor's note: Remove Table 109]

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

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

24.2.12 Concept BidirectionalIterator [iterators.bidirectional]

1 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.

2 The `BidirectionalIterator` concept refines `ForwardIterator` (24.2.11), and adds the ability to move an iterator backward as well as forward.

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

[Editor's note: Remove table 110]

3 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.

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

(4.1) — `&--a == &a`.

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

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

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

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

5 [*Note:* Bidirectional iterators allow algorithms to move iterators backward as well as forward. — *end note*]

24.2.13 Concept RandomAccessIterator [iterators.random.access]

1 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` (24.2.12) and adds support for constant-time advancement with `+=`, `+`, `-=`, and `-`, and the computation of distance in constant time with `-`. Random access iterators also support array notation via subscripting.

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

[Editor's note: Remove Table 111]

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

(2.1) — `(a += n)` is equal to `b`.

(2.2) — `&(a += n)` is equal to `&a`.

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

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

(2.5) — `a + 0` is equal to `a`.

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

(2.7) — `(b += -n)` is equal to `a`.

(2.8) — `(b -= n)` is equal to `a`.

(2.9) — `&(b -= n)` is equal to `&b`.

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

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

24.3 Indirect callable requirements [indirectcallables]

24.3.1 In general [indirectcallables.general]

1 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 Indirect callables [indirectcallables.indirectfunc]

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

```
template <class F, class... Is>
concept bool IndirectCallable() {
  return (Readable<Is>() && ...) &&
    Callable<F, value_type_t<Is>...>();
}

template <class F, class... Is>
concept bool IndirectRegularCallable() {
  return (Readable<Is>() && ...) &&
    RegularCallable<F, value_type_t<Is>...>();
}

template <class F, class... Is>
concept bool IndirectCallablePredicate() {
  return (Readable<Is>() && ...) &&
    Predicate<F, value_type_t<Is>...>();
}

template <class F, class I1, class I2 = I1>
concept bool IndirectCallableRelation() {
  return Readable<I1>() && Readable<I2>() &&
    Relation<F, value_type_t<I1>, value_type_t<I2>>();
}

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

template <class> struct indirect_result_of { };
template <class F, class... Is>
  requires IndirectCallable<remove_reference_t<F>, Is...>()
struct indirect_result_of<F(Is...)> :
  result_of<F(value_type_t<Is>...)> { };

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

24.3.3 Class template `projected` [projected]

1 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 `reference_t` of `I`.

```
template <Readable I, IndirectRegularCallable<I> Proj>
  requires RegularCallable<Proj, reference_t<I>>()
struct projected {
  using value_type = decay_t<indirect_result_of_t<Proj&(I)>>;
  result_of_t<Proj&(reference_t<I>)> operator*() const;
};
```

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

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

24.4 Common algorithm requirements [commonalgoreq]

24.4.1 In general [commonalgoreq.general]

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

2 [*Note:* The `equal_to<>` and `less<>` (20.9.5) 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 IndirectlyMovable [commonalgoreq.indirectlymovable]

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

```
template <class In>
using rvalue_reference_t =
  decltype(std::move(declval<reference_t<In>>()));

template <class In, class Out>
concept bool IndirectlyMovable() {
  return Readable<In>() &&
    Writable<Out, rvalue_reference_t<In>>();
}
```

[Editor's note: This is a simple version of `rvalue_reference_t` to be replaced with the formulation from P0022 when that proposal is integrated.]

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

```
template <class In, class Out>
concept bool IndirectlyMovableStorable() {
  return IndirectlyMovable<In, Out>() &&
    Writable<Out, value_type_t<In>>() &&
    Movable<value_type_t<In>>() &&
    Constructible<value_type_t<In>, rvalue_reference_t<In>>() &&
    Assignable<value_type_t<In>&, rvalue_reference_t<In>>();
}
```

24.4.3 Concept IndirectlyCopyable [commonalgoreq.indirectlycopyable]

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

```
template <class In, class Out>
concept bool IndirectlyCopyable() {
  return Readable<In>() &&
    Writable<Out, reference_t<In>>();
}
```

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

```
template <class In, class Out>
concept bool IndirectlyCopyableStorable() {
  return IndirectlyCopyable<In, Out>() &&
    Writable<Out, const value_type_t<In>&>() &&
    Copyable<value_type_t<In>>() &&
    Constructible<value_type_t<In>, reference_t<In>>() &&
    Assignable<value_type_t<In>&, reference_t<In>>();
}
```

24.4.4 Concept IndirectlySwappable [commonalgoreq.indirectlyswappable]

1 The `IndirectlySwappable` concept specifies a swappable relationship between the reference types of two `Readable` types.

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

24.4.5 Concept IndirectlyComparable [commonalgoreq.indirectlycomparable]

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

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

24.4.6 Concept Permutable [commonalgoreq.permutable]

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

```
template <class I>
concept bool Permutable() {
  return ForwardIterator<I>() &&
    IndirectlyMovableStorable<I, I>() &&
    IndirectlySwappable<I, I>();
}
```

24.4.7 Concept Mergeable [commonalgoreq.mergeable]

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

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

24.4.8 Concept Sortable [commonalgoreq.sortable]

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

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

24.5 Header <experimental/ranges/iterator> synopsis [iterator.synopsis]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 24.6, primitives:
  template <class Iterator> structusing 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 T> using difference_type_t
    = typename difference_type<T>::type;
  template <class T> using value_type_t
    = typename value_type<T>::type;
  template <class T> using iterator_category_t
    = typename iterator_category<T>::type;

  struct output_iterator_tag { };
  struct 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 { };

  // 24.6.5, 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 <Iterator I>
    void advance(I& i, difference_type_t<I> n);
  template <Iterator I, Sentinel<I> S>
    void advance(I& i, S bound);
  template <Iterator I, Sentinel<I> S>
    difference_type_t<I> advance(I& i, difference_type_t<I> n, S bound);
  template <Iterator I, Sentinel<I> S>
    difference_type_t<I> distance(I first, S last);
  template <Iterator I>
    I next(I x, 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, difference_type_t<I> n, S bound);
  template <BidirectionalIterator I>
    I prev(I x, difference_type_t<I> n = 1);
  template <BidirectionalIterator I>
    I prev(I x, difference_type_t<I> n, I bound);

  // 24.7, predefined iterators and sentinels:

  // 24.7.1 Reverse iterators
  template <~~class Iterator~~BidirectionalIterator I> class reverse_iterator;

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

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

template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
    requires SizedSentinel<I1, I2>()
  ~~auto~~difference_type_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_type_t<I> n,
  const reverse_iterator<~~Iterator~~I>& x);

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

// 24.7.2 Insert iterators
template <class Container> class back_insert_iterator;
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

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

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

// 24.7.3 Move iterators
template <~~class Iterator~~InputIterator I> class move_iterator;
template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
    requires EqualityComparable<I1, I2>()
  bool operator==(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
    requires EqualityComparable<I1, I2>()
  bool operator!=(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
    requires StrictTotallyOrdered<I1, I2>()
  bool operator<(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
    requires StrictTotallyOrdered<I1, I2>()
  bool operator<=(
```

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

  template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
      requires SizedSentinel<I1, I2>()
    ~~auto~~difference_type_t<I2> operator-(
      const move_iterator<~~Iterator1~~I1>& x,
      const move_iterator<~~Iterator2~~I2>& y)~~ ->decltype(y.base() - x.base())~~;
  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);
  template <~~class Iterator~~InputIterator I>
    move_iterator<~~Iterator~~I> make_move_iterator(~~Iterator~~I i);

  template <Semiregular S> class move_sentinel;

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

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

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

  // 24.7.4 Common iterators

  template <Iterator I, Sentinel<I> S>
    requires !Same<I, S>()
  class common_iterator;

  template <Readable I, class S>
```

```
struct value_type<common_iterator<I, S>>;

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

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

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

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

// 24.7.5 Default sentinels
class default_sentinel;
// 24.7.6 Counted iterators
template <Iterator I> class counted_iterator;

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

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

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

// 24.7.8 Unreachable sentinels
class unreachable;
template <Iterator I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator==(unreachable, const I&) noexcept;
template <Iterator I>
  constexpr bool operator!=(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator!=(unreachable, const I&) noexcept;

// 24.7.7 Dangling wrapper
template <class T> class dangling;

// 24.8, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(default_sentinel x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
          default_sentinel y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
```

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

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

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

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

// 24.10, 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 <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
  -> decltype(std::begin(c));
template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
  -> decltype(std::end(c));
template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());
template <class C> auto rend(C& c) -> decltype(c.rend());
template <class C> auto rend(const C& c) -> decltype(c.rend());
template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);
template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);
template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);
template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);
template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));
template <class C> auto crend(const C& c) -> decltype(std::rend(c));
```

```
  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 ;
  }

  // 24.11, Range primitives:
  namespace {
    constexpr unspecified  size = unspecified ;
    constexpr unspecified  empty = unspecified ;
    constexpr unspecified  data = unspecified ;
    constexpr unspecified  cdata = unspecified ;
  }
  template <Range R>
  difference_type_t<iterator_t<R>> distance(R&& r);
  template <SizedRange R>
  difference_type_t<iterator_t<R>> distance(R&& r);

}}}}

namespace std {
  // 24.6.2, iterator traits
  template <experimental::ranges::Iterator I>
    struct iterator_traits;
  template <experimental::ranges::InputIterator I>
    struct iterator_traits;
  template <experimental::ranges::InputIterator I>
      requires Sentinel<I, I>()
    struct iterator_traits;
}
```

1 Any entities declared or defined in namespace `std` in header `<iterator>` that are not already defined in namespace `std::experimental::ranges` in header `<experimental/ranges/iterator>` are imported with *using-declarations* (7.3.3). [Editor's note: There are no such entities, but there may be in the future.]

24.6 Iterator primitives [iterator.primitives]

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

24.6.1 Iterator ~~traits~~associated types [iterator.assoc]

1 To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if ~~`Iterator` is the type of an iterator~~ `WI` is the name of a type that satisfies the `WeaklyIncrementable` concept (24.2.4), `R` is the name of a type that satisfies the `Readable` concept (24.2.2), and `II` is the name of a type that satisfies the `InputIterator` concept (24.2.9) concept, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
```

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

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

```
reference_t<R>
```

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

```
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

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

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

may be defined as `void`.

2 The template `iterator_traits<Iterator>` is defined as

```
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

```
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

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

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

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

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

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

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

[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`. `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 `iterator_category_t<T>` is implemented as if:

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

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

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

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

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

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

8 If `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::random_access_iterator_tag`.

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

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

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

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

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

9 [*Note:* If there is an additional pointer type `__far` such that the difference of two `__far` is of type `long`, an implementation may define

```
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 document specifies the existence of an `iterator_traits` alias that collects an iterator's associated types. It is defined as if:

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

```
    using reference = reference_t<I>;
    using pointer = typename __pointer_type<I>::type;
    using iterator_category = iterator_category_t<I>;
  };
  template <class I>
    using iterator_traits = __iterator_traits<I>;
```

11 [*Note:* `iterator_traits` is an alias template 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(BidirectionalIteratorI first, BidirectionalIteratorI last) {
  typename iterator_traits<BidirectionalIterator>::difference_typedifference_type_t<I> n =
    distance(first, last);
  --n;
  while(n > 0) {
    typename iterator_traits<BidirectionalIterator>::value_typevalue_type_t<I>
      tmp = *first;
    *first++ = *--last;
    *last = tmp;
    n -= 2;
  }
}
```

— *end example*]

24.6.2 Standard iterator traits [iterator.stdtraits]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

24.6.3 Basic iterator [iterator.basic]

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

```
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;
  };
}
```

24.6.4 Standard iterator tags [std.iterator.tags]

1 It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which ~~are~~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*] ~~They~~The category tags are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every input iterator of type `I`~~`terator`~~, ~~`iterator_traits<Iterator>::iterator_category`~~ `iterator_category_t<I>` shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  struct output_iterator_tag { };
  struct 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 { };
}}}}
```

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

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

```
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;
};

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

```
    using type = bidirectional_iterator_tag;
  };
```

~~Typically, however, it would be easier to derive `BinaryTreeIterator<T>` from `iterator<bidirectional_iterator_tag, T, ptrdiff_t, T*, T&>` .~~ — *end example*]

4 [*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
}
```

— *end example*]

5 [*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.
};
```

6 Then there is no need to specialize the `iterator_traits` template. — *end example*]

24.6.5 Iterator operations [iterator.operations]

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

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

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

2 *Requires:* `n` shall be negative only for bidirectional ~~and random access~~ iterators.

3 *Effects:* For random access iterators, equivalent to `i += n`. Otherwise, i~~I~~ncrements (or decrements for negative `n`) iterator ~~reference~~ `i` by `n`.

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

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

5 *Effects:*

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

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

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

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

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

7 *Effects:*

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

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

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

(7.2) — Otherwise, increments (or decrements for negative `n`) iterator `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);
```

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

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

10 *Effects:* If ~~`InputIterator` meets the requirements of random access iterator~~ `SizedSentinel<S, I>()` 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 `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);
```

```
template <Iterator I>
  I next(I x, difference_type_t<I> n = 1);
```

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

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

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

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

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

```
template <class BidirectionalIterator>
  BidirectionalIterator prev(BidirectionalIterator x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

```
template <BidirectionalIterator I>
  I prev(I x, difference_type_t<I> n = 1);
```

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

```
template <BidirectionalIterator I>
  I prev(I x, difference_type_t<I> n, I bound);
```

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

24.7 Iterator adaptors [iterators.predef]

24.7.1 Reverse iterators [iterators.reverse]

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

24.7.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <~~class Iterator~~BidirectionalIterator I>
  class reverse_iterator ~~: public~~{

      iterator<typename iterator_traits<Iterator>::iterator_category,
      typename iterator_traits<Iterator>::value_type,
      typename iterator_traits<Iterator>::difference_type,
      typename iterator_traits<Iterator>::pointer,
      typename iterator_traits<Iterator>::reference> {

  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 = difference_type_t<I>;
    using value_type = value_type_t<I>;
    using iterator_category = iterator_category_t<I>;
    using reference = reference_t<I>;
    using pointer = I;

    reverse_iterator();
    explicit reverse_iterator(~~Iterator~~I x);
    ~~template <class U>~~
    reverse_iterator(const reverse_iterator<~~U~~ConvertibleTo<I>>& ~~u~~i);
```

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

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

```
  template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
      requires StrictTotallyOrdered<I1, I2>()
    bool operator<=(
      const reverse_iterator<~~Iterator1~~I1>& x,
      const reverse_iterator<~~Iterator2~~I2>& y);
  template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
      requires SizedSentinel<I1, I2>()
    ~~auto~~difference_type_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_type_t<I> n,
    const reverse_iterator<~~Iterator~~I>& x);

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

~~24.7.1.2 reverse_iterator requirements [reverse.iter.requirements]~~

1 ~~The template parameter `Iterator` shall meet all the requirements of a Bidirectional Iterator (24.2.12).~~

2 ~~Additionally, `Iterator` shall meet the requirements of a Random Access Iterator (24.2.13) if any of the members `operator+` (24.7.1.3.8), `operator-` (24.7.1.3.10), `operator+=` (24.7.1.3.9), `operator-=` (24.7.1.3.11), `operator []` (24.7.1.3.12), or the global operators `operator<` (24.7.1.3.15), `operator>` (24.7.1.3.16), `operator <=` (24.7.1.3.18), `operator>=` (24.7.1.3.17), `operator-` (24.7.1.3.19) or `operator+` (24.7.1.3.20) are referenced in a way that requires instantiation (14.7.1).~~

24.7.1.3 `reverse_iterator` operations [reverse.iter.ops]

24.7.1.3.1 `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`~~`I`.

```
explicit reverse_iterator(~~Iterator~~I x);
```

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

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

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

24.7.1.3.2 `reverse_iterator::operator=` [reverse.iter.op=]

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

1 *Effects:* Assigns ~~`u.base()`~~`i.current` to `current`.

2 *Returns:* `*this`.

24.7.1.3.3 Conversion [reverse.iter.conv]

~~Iterator~~I base() const; ~~// explicit~~

1 *Returns:* current.

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

reference operator*() const;

1 *Effects:* Equivalent to *prev(current)

~~Iterator tmp = current;~~
~~return *--tmp;~~

24.7.1.3.5 operator-> [reverse.iter.opref]

pointer operator->() const;

1 ~~*Returns:*~~ *Effects:* Equivalent to ~~std::addressof(operator*())~~prev(current).

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

reverse_iterator& operator++();

1 *Effects:* --current;

2 *Returns:* *this.

reverse_iterator operator++(int);

3 *Effects:*

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

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

reverse_iterator& operator--();

1 *Effects:* ++current

2 *Returns:* *this.

reverse_iterator operator--(int);

3 *Effects:*

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

24.7.1.3.8 operator+ [reverse.iter.op+]

reverse_iterator
  operator+(~~typename reverse_iterator<Iterator>::~~difference_type n) const~~;~~
    requires RandomAccessIterator<I>();

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

24.7.1.3.9 `operator+=` [reverse.iter.op+=]

`reverse_iterator&`
`operator+=(`~~`typename reverse_iterator<Iterator>::`~~`difference_type n)`~~`;`~~
`requires RandomAccessIterator<I>();`

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

2 *Returns:* `*this`.

24.7.1.3.10 `operator-` [reverse.iter.op-]

`reverse_iterator`
`operator-(`~~`typename reverse_iterator<Iterator>::`~~`difference_type n) const`~~`;`~~
`requires RandomAccessIterator<I>();`

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

24.7.1.3.11 `operator-=` [reverse.iter.op-=]

`reverse_iterator&`
`operator-=(`~~`typename reverse_iterator<Iterator>::`~~`difference_type n)`~~`;`~~
`requires RandomAccessIterator<I>();`

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

2 *Returns:* `*this`.

24.7.1.3.12 `operator[]` [reverse.iter.opindex]

~~*`unspecified`*~~ `reference operator[](`
~~`typename reverse_iterator<Iterator>::`~~`difference_type n) const`~~`;`~~
`requires RandomAccessIterator<I>();`

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

24.7.1.3.13 `operator==` [reverse.iter.op==]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.current == y.current`.

24.7.1.3.14 `operator!=` [reverse.iter.op!=]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.current != y.current`.

24.7.1.3.15 `operator<` [reverse.iter.op<]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.current > y.current`.

24.7.1.3.16 `operator>` [reverse.iter.op>]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.current < y.current`.

24.7.1.3.17 `operator>=` [reverse.iter.op>=]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.current <= y.current`.

24.7.1.3.18 `operator<=` [reverse.iter.op<=]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.current >= y.current`.

24.7.1.3.19 `operator-` [reverse.iter.opdiff]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `y.current - x.current`.

24.7.1.3.20 `operator+` [reverse.iter.opsum]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `reverse_iterator<~~Iterator~~I>(x.current - n)`.

24.7.1.3.21 Non-member function `make_reverse_iterator()` [reverse.iter.make]

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

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

24.7.2 Insert iterators [iterators.insert]

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

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

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

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

24.7.2.1 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?]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Container>
  class back_insert_iterator ~~:~~{
    ~~public iterator<output_iterator_tag, void, void, void, void> {~~
  ~~protected~~private:
    Container* container; // exposition only

  public:
    ~~typedef Container~~using container_type = Container;
    using difference_type = ptrdiff_t;
    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*();
    back_insert_iterator~~<Container>~~& operator++();
    back_insert_iterator~~<Container>~~ operator++(int);
  };

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

24.7.2.2 `back_insert_iterator` operations [back.insert.iter.ops]

24.7.2.2.1 `back_insert_iterator` constructor [back.insert.iter.cons]

`constexpr back_insert_iterator();`

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

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

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

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

```
back_insert_iterator~~<Container>~~&
  operator=(const typename Container::value_type& value);
```

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

2 *Returns:* `*this`.

```
back_insert_iterator~~<Container>~~&
  operator=(typename Container::value_type&& value);
```

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

4 *Returns:* `*this`.

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

```
back_insert_iterator~~<Container>~~& operator*();
```

1 *Returns:* `*this`.

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

```
back_insert_iterator~~<Container>~~& operator++();
back_insert_iterator~~<Container>~~ operator++(int);
```

1 *Returns:* `*this`.

24.7.2.2.5 `back_inserter` [back.inserter]

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

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

24.7.2.3 Class template `front_insert_iterator` [front.insert.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class Container>
  class front_insert_iterator ~~:~~{
    ~~public iterator<output_iterator_tag, void, void, void, void> {~~
  ~~protected~~private:
    Container* container; // exposition only

  public:
    ~~typedef Container~~using container_type = Container;
    using difference_type = ptrdiff_t;
    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);
}}}}
```

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

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

```
constexpr front_insert_iterator();
```

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

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

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

24.7.2.4.2 front_insert_iterator::operator= [front.insert.iter.op=]

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

24.7.2.4.3 front_insert_iterator::operator* [front.insert.iter.op*]

```
front_insert_iterator~~<Container>~~& operator*();
```

1 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

24.7.2.4.5 front_inserter [front.inserter]

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

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

24.7.2.5 Class template insert_iterator [insert.iterator]

```
namespace std { namespace experimental { namespace ranges { 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 Containerusing container_type = Container;
    using difference_type = ptrdiff_t;
    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);
}}}}
```

24.7.2.6 insert_iterator operations [insert.iter.ops]

24.7.2.6.1 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.

24.7.2.6.2 insert_iterator::operator= [insert.iter.op=]

```
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 *Effects:* Equivalent to

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

4 *Returns:* *this.

24.7.2.6.3 insert_iterator::operator* [insert.iter.op*]

```
insert_iterator<Container>& operator*();
```

1 *Returns:* *this.

24.7.2.6.4 `insert_iterator::operator++` [insert.iter.op++]

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

1 *Returns:* `*this`.

24.7.2.6.5 `inserter` [inserter]

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

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

24.7.3 Move iterators [iterators.move]

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

2 [*Example:*

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

— *end example*]

3 Class template `move_sentinel` is a sentinel adaptor useful for denoting ranges together with `move_iterator`. When an input iterator type `I` and sentinel type `S` satisfy `Sentinel<S, I>()`, `Sentinel<move_sentinel<S>, move_iterator<I>>()` is satisfied as well.

4 [*Example:* A `move_if` algorithm is easily implemented with `copy_if` using `move_iterator` and `move_sentinel`:

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
          IndirectCallablePredicate<I> Pred>
  requires IndirectlyMovable<I, O>()
void move_if(I first, S last, O out, Pred pred)
{
  copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}
```

— *end example*]

24.7.3.1 Class template `move_iterator` [move.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class IteratorInputIterator 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   = difference_type_t<I>;
    using value_type        = value_type_t<I>;
    using iterator_category = input_iterator_tag;
    using reference         = see below;


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

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

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

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

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

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

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

  template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
      requires SizedSentinel<I1, I2>()
    ~~auto~~difference_type_t<I2> operator-(
      const move_iterator<~~Iterator1~~I1>& x,
      const move_iterator<~~Iterator2~~I2>& y)~~ ->decltype(y.base() - x.base())~~;
  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);
  template <~~class Iterator~~InputIterator I>
    move_iterator<~~Iterator~~I> make_move_iterator(~~Iterator~~I i);
}}}}
```

1 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*.

2 [*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*]

~~24.7.3.2 move_iterator requirements [move.iter.requirements]~~

1 ~~The template parameter `Iterator` shall meet the requirements for an Input Iterator (24.2.9). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.2.12) or a Random Access Iterator (24.2.13), respectively.~~

24.7.3.3 `move_iterator` operations [move.iter.ops]

24.7.3.3.1 `move_iterator` constructors [move.iter.op.const]

```
move_iterator();
```

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

```
explicit move_iterator(~~Iterator~~I i);
```

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

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

3 *Effects:* Constructs a `move_iterator`, initializing `current` with ~~`u.base()`~~`i.current`.

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

24.7.3.3.2 `move_iterator::operator=` [move.iter.op=]

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

1 *Effects:* Assigns ~~`u.base()`~~`i.current` to `current`.

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

24.7.3.3.3 `move_iterator` conversion [move.iter.op.conv]

~~`Iterator`~~`I base() const;`

1 *Returns:* `current`.

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

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

1 ~~*Returns:*~~ *Effects:* Equivalent to ~~`std::move`~~`static_cast<reference>(*current)`.

24.7.3.3.5 `move_iterator::operator->` [move.iter.op.ref]

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

1 *Returns:* `current`.

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

```
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;
```

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

`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;
```

24.7.3.3.8 `move_iterator::operator+` [move.iter.op.+]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `move_iterator(current + n)`.

24.7.3.3.9 `move_iterator::operator+=` [move.iter.op.+=]

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

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

2 *Returns:* `*this`.

24.7.3.3.10 `move_iterator::operator-` [move.iter.op.-]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `move_iterator(current - n)`.

24.7.3.3.11 `move_iterator::operator-=` [move.iter.op.-=]

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

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

2 *Returns:* `*this`.

24.7.3.3.12 `move_iterator::operator[]` [move.iter.op.index]

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

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

24.7.3.3.13 `move_iterator` comparisons [move.iter.op.comp]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `x.`~~`base()`~~`current == y.`~~`base()`~~`current`.

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

2 ~~*Returns:*~~ *Effects:* Equivalent to `!(x == y)`.

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

3 ~~*Returns:*~~ *Effects:* Equivalent to `x.`~~`base()`~~`current < y.`~~`base()`~~`current`.

```
template <class ~~Iterator1~~I1, class ~~Iterator2~~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~~I1, class ~~Iterator2~~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~~I1, class ~~Iterator2~~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)`.

24.7.3.3.14 `move_iterator` non-member functions [move.iter.nonmember]

```
template <class ~~Iterator1~~I1, class ~~Iterator2~~I2>
    requires SizedSentinel<I1, I2>()
  ~~auto~~difference_type_t<I2> operator-(
    const move_iterator<~~Iterator1~~I1>& x,
    const move_iterator<~~Iterator2~~I2>& y)~~ ->decltype(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~~InputIterator I>
  move_iterator<~~Iterator~~I> make_move_iterator(~~Iterator~~I i);
```

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

24.7.3.4 Class template `move_sentinel` [move.sentinel]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <Semiregular S>
  class move_sentinel {
  public:
    constexpr move_sentinel();
    explicit move_sentinel(S s);
    move_sentinel(const move_sentinel<ConvertibleTo<S>>& s);
    move_sentinel& operator=(const move_sentinel<ConvertibleTo<S>>& s);

    S base() const;

  private:
    S last; // exposition only
```

```
  };

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

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

  template <Semiregular S>
    move_sentinel<S> make_move_sentinel(S s);
}}}}
```

24.7.3.5 `move_sentinel` operations [move.sent.ops]

24.7.3.5.1 `move_sentinel` constructors [move.sent.op.const]

```
constexpr move_sentinel();
```

1 *Effects:* Constructs a `move_sentinel`, value-initializing `last`. If `S` is a literal type, this constructor shall be a `constexpr` constructor.

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

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

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

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

24.7.3.5.2 `move_sentinel::operator=` [move.sent.op=]

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

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

24.7.3.5.3 `move_sentinel` comparisons [move.sent.op.comp]

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

1 *Effects:* Equivalent to `i.current == s.last`.

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

2 *Effects:* Equivalent to `!(i == s)`.

24.7.3.5.4 `move_sentinel` non-member functions [move.sent.nonmember]

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

1 *Effects:* Equivalent to `s.last - i.current`.

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

2 *Effects:* Equivalent to `i.current - s.last`.

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

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

24.7.4 Common iterators [iterators.common]

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

2 [*Note:* The `common_iterator` type is useful for interfacing with legacy code that expects the begin and end of a range to have the same type. — *end note*]

3 [*Example:*

```
template <class ForwardIterator>
void fun(ForwardIterator begin, ForwardIterator end);

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

— *end example*]

24.7.4.1 Class template `common_iterator` [common.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <Iterator I, Sentinel<I> S>
    requires !Same<I, S>()
  class common_iterator {
```

```
  public:
    using difference_type = difference_type_t<I>;

    common_iterator();
    common_iterator(I i);
    common_iterator(S s);
    common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
    common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);

    ~common_iterator();

    see below operator*();
    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 <Readable I, class S>
  struct value_type<common_iterator<I, S>> {
    using type = value_type_t<I>;
  };

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

  template <ForwardIterator I, class S>
  struct iterator_category<common_iterator<I, S>> {
    using type = forward_iterator_tag;
  };

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

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

1 [*Note:* It is unspecified whether `common_iterator`'s members `iter` and `sent` have distinct addresses or not. — *end note*]

24.7.4.2 `common_iterator` operations [common.iter.ops]

24.7.4.2.1 `common_iterator` constructors [common.iter.op.const]

```
common_iterator();
```

1 *Effects:* Constructs a `common_iterator`, value-initializing `is_sentinel` and `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`.

2 *Remarks:* It is unspecified whether any initialization is performed for `sent`.

```
common_iterator(I i);
```

3 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `false` and `iter` with `i`.

4 *Remarks:* It is unspecified whether any initialization is performed for `sent`.

```
common_iterator(S s);
```

5 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `true` and `sent` with `s`.

6 *Remarks:* It is unspecified whether any initialization is performed for `iter`.

```
common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
```

7 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `u.is_sentinel`.

(7.1) — If `u.is_sentinel` is `true`, `sent` is initialized with `u.sent`.

(7.2) — If `u.is_sentinel` is `false`, `iter` is initialized with `u.iter`.

Remarks:

(7.3) — If `u.is_sentinel` is `true`, it is unspecified whether any initialization is performed for `iter`.

(7.4) — If `u.is_sentinel` is `false`, it is unspecified whether any initialization is performed for `sent`.

24.7.4.2.2 `common_iterator::operator=` [common.iter.op=]

```
common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
```

1 *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`.

2 *Returns:* `*this`

```
~common_iterator();
```

3 *Effects:* Destroys any members that are currently initialized.

24.7.4.2.3 `common_iterator::operator*` [common.iter.op.star]

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

1 *Requires:* `!is_sentinel`

2 *Effects:* Equivalent to `*iter`.

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

3 *Requires:* `!is_sentinel`

4 *Effects:* Given an object `i` of type `I`

(4.1) — if `I` is a pointer type or if the expression `i.operator->()` is well-formed, this function returns `iter`.

(4.2) — Otherwise, if the expression `*iter` is a 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_;
  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`.

24.7.4.2.4 `common_iterator::operator++` [common.iter.op.incr]

```
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;
```

24.7.4.2.5 `common_iterator` comparisons [common.iter.op.comp]

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

1 *Effects:* Equivalent to

```
x.is_sentinel ?
    (y.is_sentinel || y.iter == x.sent) :
    (!y.is_sentinel || x.iter == y.sent)
```

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

2 *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 <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
bool operator!=(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

3 *Effects:* Equivalent to `!(x == y)`.

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

4 *Effects:* Equivalent to

```
x.is_sentinel ?
    (y.is_sentinel ? 0 : x.sent - y.iter) :
    (y.is_sentinel ?
         x.iter - y.sent :
         x.iter - y.iter);
```

24.7.5 Default sentinels [default.sentinels]

24.7.5.1 Class `default_sentinel` [default.sent]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  class default_sentinel { };
}}}}
```

1 Class `default_sentinel` is an empty type used to denote the end of a range. It is intended to be used together with iterator types that know the bound of their range (e.g., `counted_iterator` (24.7.6.1)).

24.7.6 Counted iterators [iterators.counted]

1 Class template `counted_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of its distance from its starting position. It can be used together with class `default_sentinel` in calls to generic algorithms to operate on a range of N elements starting at a given position without needing to know the end position *a priori*.

2 [*Example:*

```
list<string> s;
// populate the list s with at least 10 strings
vector<string> v(make_counted_iterator(s.begin(), 10),
                 default_sentinel()); // copies 10 strings into v
```

— *end example*]

3 Two values `i1` and `i2` of (possibly differing) types `counted_iterator<I1>` and `counted_iterator<I2>` refer to elements of the same sequence if and only if `next(i1.base(), i1.count())` and `next(i2.base(), i2.count())` refer to the same (possibly past-the-end) element.

24.7.6.1 Class template counted_iterator [counted.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <Iterator I>
  class counted_iterator {
  public:
    using iterator_type = I;
    using difference_type = difference_type_t<I>;

    counted_iterator();
    counted_iterator(I x, difference_type_t<I> n);
    counted_iterator(const counted_iterator<ConvertibleTo<I>>& i);
    counted_iterator& operator=(const counted_iterator<ConvertibleTo<I>>& i);

    I base() const;
    difference_type_t<I> count() const;
    see below operator*();
    see below operator*() 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>();
    see below operator[](difference_type n) const
      requires RandomAccessIterator<I>();
  private:
    I current; // exposition only
    difference_type_t<I> cnt; // exposition only
  };

  template <Readable I>
  struct value_type<counted_iterator<I>> {
    using type = value_type_t<I>;
  };

  template <InputIterator I>
  struct iterator_category<counted_iterator<I>> {
    using type = iterator_category_t<I>;
  };
```

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

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

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

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

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

template <Iterator I>
  void advance(counted_iterator<I>& i, difference_type_t<I> n);
```

```
}}}}
```

24.7.6.2 `counted_iterator` operations [counted.iter.ops]

24.7.6.2.1 `counted_iterator` constructors [counted.iter.op.const]

```
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, difference_type_t<I> n);
```

2 *Requires:* `n >= 0`

3 *Effects:* Constructs a `counted_iterator`, initializing `current` with `i` and `cnt` with `n`.

```
counted_iterator(const counted_iterator<ConvertibleTo<I>>& i);
```

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

24.7.6.2.2 `counted_iterator::operator=` [counted.iter.op=]

```
counted_iterator& operator=(const counted_iterator<ConvertibleTo<I>>& i);
```

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

24.7.6.2.3 `counted_iterator` conversion [counted.iter.op.conv]

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

1 *Returns:* `current`.

24.7.6.2.4 `counted_iterator` count [counted.iter.op.cnt]

```
difference_type_t<I> count() const;
```

1 *Returns:* `cnt`.

24.7.6.2.5 `counted_iterator::operator*` [counted.iter.op.star]

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

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

24.7.6.2.6 `counted_iterator::operator++` [counted.iter.op.incr]

```
counted_iterator& operator++();
```

1 *Requires:* `cnt > 0`

2 *Effects:* Equivalent to

```
++current;
--cnt;
```

3 *Returns:* `*this`.

```
counted_iterator operator++(int);
```

4 *Requires:* `cnt > 0`

5 *Effects:* Equivalent to

```
counted_iterator tmp = *this;
++current;
--cnt;
return tmp;
```

24.7.6.2.7 `counted_iterator::operator--` [counted.iter.op.decr]

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

1 *Effects:* Equivalent to

```
--current;
++cnt;
```

2 *Returns:* `*this`.

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

3 *Effects:* Equivalent to

```
counted_iterator tmp = *this;
--current;
++cnt;
return tmp;
```

24.7.6.2.8 `counted_iterator::operator+` [counted.iter.op.+]

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

1 *Requires:* `n <= cnt`

2 *Effects:* Equivalent to `counted_iterator(current + n, cnt - n)`.

24.7.6.2.9 `counted_iterator::operator+=` [counted.iter.op.+=]

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

1 *Requires:* `n <= cnt`

2 *Effects:*

```
current += n;
cnt -= n;
```

3 *Returns:* `*this`.

24.7.6.2.10 `counted_iterator::operator-` [counted.iter.op.-]

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

1 *Requires:* `-n <= cnt`

2 *Effects:* Equivalent to `counted_iterator(current - n, cnt + n)`.

24.7.6.2.11 `counted_iterator::operator-=` [counted.iter.op.-=]

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

1 *Requires:* `-n <= cnt`

2 *Effects:*

```
current -= n;
cnt += n;
```

3 *Returns:* `*this`.

24.7.6.2.12 `counted_iterator::operator[]` [counted.iter.op.index]

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

1 *Requires:* `n <= cnt`

2 *Effects:* Equivalent to `current[n]`.

24.7.6.2.13 `counted_iterator` comparisons [counted.iter.op.comp]

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

1 *Requires:* `x` and `y` shall refer to elements of the same sequence (24.7.6).

2 *Effects:* Equivalent to `x.cnt == y.cnt`.

```
bool operator==(
  const counted_iterator<auto>& x, default_sentinel);
bool operator==(
  default_sentinel, const counted_iterator<auto>& x);
```

3 *Effects:* Equivalent to `x.cnt == 0`.

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

4 *Requires:* For the first overload, `x` and `y` shall refer to elements of the same sequence (24.7.6).

5 *Effects:* Equivalent to `!(x == y)`.

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

6 *Requires:* `x` and `y` shall refer to elements of the same sequence (24.7.6).

7 *Effects:* Equivalent to `y.cnt < x.cnt`.

Remark: The argument order in the *Effects* clause is reversed because `cnt` counts down, not up.

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

8 *Requires:* `x` and `y` shall refer to elements of the same sequence (24.7.6).

9 *Effects:* Equivalent to `!(y < x)`.

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

10 *Requires:* `x` and `y` shall refer to elements of the same sequence (24.7.6).

11 *Effects:* Equivalent to `y < x`.

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

12 *Requires:* `x` and `y` shall refer to elements of the same sequence (24.7.6).

13 *Effects:* Equivalent to `!(x < y)`.

24.7.6.2.14 `counted_iterator` non-member functions [counted.iter.nonmember]

```
  template <class I1, class I2>
      requires Common<I1, I2>()
  difference_type_t<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Requires:* `x` and `y` shall refer to elements of the same sequence (24.7.6).

2 *Effects:* Equivalent to `y.cnt - x.cnt`.

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

3 *Effects:* Equivalent to `-x.cnt`.

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

4 *Effects:* Equivalent to `y.cnt`.

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

5 *Requires:* `n <= x.cnt`.

6 *Effects:* Equivalent to `x + n`.

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

7 *Requires:* `n >= 0`.

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

```
template <Iterator I>
  void advance(counted_iterator<I>& i, difference_type_t<I> n);
```

9 *Requires:* `n <= i.cnt`.

10 *Effects:*

```
i = make_counted_iterator(next(i.current, n), i.cnt - n);
```

24.7.7 Dangling wrapper [dangling.wrappers]

24.7.7.1 Class template `dangling` [dangling.wrap]

1 Class template `dangling` is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.

```
namespace std { namespace experimental { namespace ranges { 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,
      iterator_t<R>,
      dangling<iterator_t<R>>>;
}}}}
```

24.7.7.2 `dangling` operations [dangling.wrap.ops]

24.7.7.2.1 `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`.

24.7.7.2.2 `dangling::get_unsafe` [dangling.wrap.op.get]

```
T get_unsafe() const;
```

1 *Returns:* `value`.

24.7.8 Unreachable sentinel [unreachable.sentinels]

24.7.8.1 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 { inline namespace v1 {
  class unreachable { };
  template <Iterator I>
    constexpr bool operator==(const I&, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator==(unreachable, const I&) noexcept;
  template <Iterator I>
    constexpr bool operator!=(const I&, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator!=(unreachable, const I&) noexcept;
}}}}
```

24.7.8.2 `unreachable` operations [unreachable.sentinel.ops]

24.7.8.2.1 `operator==` [unreachable.sentinel.op==]

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

1 *Returns:* `false`.

24.7.8.2.2 `operator!=` [unreachable.sentinel.op!=]

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

1 *Returns:* `true`.

24.8 Stream iterators [iterators.stream]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[*Example:*

```
partial_sum(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. — *end example*]

24.8.1 Class template `istream_iterator` [istream.iterator]

1 The class template `istream_iterator` is an input iterator (24.2.9) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on

the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

2 Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class T, class charT = char, class traits = char_traits<charT>,
      class Distance = ptrdiff_t>
  class istream_iterator:
    public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
  public:
    typedef input_iterator_tag iterator_category;
    typedef Distance difference_type;
    typedef T value_type;
    typedef const T& reference;
    typedef const T* pointer;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT, traits> istream_type;
    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);
  template <class T, class charT, class traits, class Distance>
    bool operator==(default_sentinel x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
            default_sentinel y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(default_sentinel x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
```

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

24.8.1.1 istream_iterator constructors and destructor [istream.iterator.cons]

```
see below istream_iterator();
see below istream_iterator(default_sentinel);
```

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then ~~this~~these constructors shall be ~~a~~ `constexpr` constructors.

2 *Postcondition:* `in_stream == ~~0~~nullptr`.

```
istream_iterator(istream_type& s);
```

3 *Effects:* Initializes `in_stream` with `&s`. `value` may be initialized during construction or the first time it is referenced.

4 *Postcondition:* `in_stream == &s`.

```
istream_iterator(const istream_iterator& x) = default;
```

5 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6 *Postcondition:* `in_stream == x.in_stream`.

```
~istream_iterator() = default;
```

7 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

24.8.1.2 istream_iterator operations [istream.iterator.ops]

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

1 *Returns:* `value`.

```
const T* operator->() const;
```

2 *Effects:* Equivalent to ~~`&(operator*())`~~`std::addressof(operator*())`.

```
istream_iterator~~<T, charT, traits, Distance>~~& operator++();
```

3 *Requires:* `in_stream != ~~0~~nullptr`.

4 *Effects:* `*in_stream >> value`.

5 *Returns:* `*this`.

```
istream_iterator~~<T, charT, traits, Distance>~~ operator++(int);
```

6 *Requires:* `in_stream != ~~0~~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)`

24.8.2 Class template ostream_iterator [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 { 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 ptrdiff_t difference_type;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT, traits> ostream_type;
    constexpr ostream_iterator() noexcept;
    ostream_iterator(ostream_type& s) noexcept;
    ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;
    ostream_iterator(const ostream_iterator~~<T, charT, traits>~~& x) 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
  };
}}}}
```

24.8.2.1 ostream_iterator constructors and destructor [ostream.iterator.cons.des]

`constexpr ostream_iterator() noexcept;`

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

`ostream_iterator(ostream_type& s) noexcept;`

2 *Effects:* Initializes `out_stream` with `&s` and `delim` with ~~null~~`nullptr`.

`ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;`

3 *Effects:* Initializes `out_stream` with `&s` and `delim` with `delimiter`.

`ostream_iterator(const ostream_iterator& x) noexcept;`

4 *Effects:* Constructs a copy of `x`.

`~ostream_iterator();`

5 *Effects:* The iterator is destroyed.

24.8.2.2 ostream_iterator operations [ostream.iterator.ops]

`ostream_iterator& operator=(const T& value);`

1 *Effects:* Equivalent to:

```
*out_stream << value;
if(delim != ~~0~~nullptr)
  *out_stream << delim;
return ~~(~~*this~~)~~;
```

`ostream_iterator& operator*();`

2 *Returns:* `*this`.

```
ostream_iterator& operator++();
ostream_iterator~~&~~ operator++(int);
```

3 *Returns:* `*this`.

24.8.3 Class template istreambuf_iterator [istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (24.2.9) 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(~~0~~nullptr)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_-type` value is returned. It is impossible to assign a character via an input iterator.

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

    class proxy;                           // exposition only

    constexpr istreambuf_iterator() noexcept;
    constexpr istreambuf_iterator(default_sentinel) noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
    ~istreambuf_iterator() = default;
    istreambuf_iterator(istream_type& s) noexcept;
    istreambuf_iterator(streambuf_type* s) noexcept;
    istreambuf_iterator(const proxy& p) noexcept;
    charT operator*() const;
    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);
}}}}
```

24.8.3.1 Class template `istreambuf_iterator::proxy` [istreambuf.iterator::proxy]

```
namespace std { namespace experimental { namespace ranges { 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.

24.8.3.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

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

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

```
istreambuf_iterator(basic_istream<charT, traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT, traits>* s) noexcept;
```

2 *Effects:* Constructs an `istreambuf_iterator<>` that uses the `basic_streambuf<>` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

```
istreambuf_iterator(const proxy& p) noexcept;
```

3 *Effects:* Constructs a `istreambuf_iterator<>` that uses the `basic_streambuf<>` object pointed to by the `proxy` object's constructor argument `p`.

24.8.3.3 `istreambuf_iterator::operator*` [istreambuf.iterator::op*]

```
charT operator*() const
```

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

24.8.3.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```
istreambuf_iterator~~<charT, traits>~~&
    istreambuf_iterator<charT, traits>::operator++();
```

1 *Effects:* Equivalent to `sbuf_->sbumpc()`.

2 *Returns:* `*this`.

```
proxy istreambuf_iterator<charT, traits>::operator++(int);
```

3 *Effects:* Equivalent to `proxy(sbuf_->sbumpc(), sbuf_)`.

24.8.3.5 `istreambuf_iterator::equal` [istreambuf.iterator::equal]

```
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.

24.8.3.6 operator== [istreambuf.iterator::op==]

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

1 ~~*Returns:*~~ *Effects:* Equivalent to `a.equal(b)`.

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

2 *Effects:* Equivalent to `istreambuf_iterator<charT, traits>{}.equal(b)`.

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

3 *Effects:* Equivalent to `a.equal(istreambuf_iterator<charT, traits>{})`.

24.8.3.7 operator!= [istreambuf.iterator::op!=]

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

1 ~~*Returns:* !a.equal(b).~~ *Effects:* Equivalent to `!(a == b)`

24.8.4 Class template ostreambuf_iterator [ostreambuf.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator :
    public iterator<output_iterator_tag, void, void, void, void> {
  public:
    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
  };
}}}}
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

24.8.4.1 ostreambuf_iterator constructors [ostreambuf.iter.cons]

`constexpr ostreambuf_iterator() noexcept;`

1 *Effects:* Initializes `sbuf_` with ~~null~~`nullptr`.

`ostreambuf_iterator(ostream_type& s) noexcept;`

2 *Requires:* `s.rdbuf() != nullptr` ~~shall not null pointer~~.

3 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

`ostreambuf_iterator(streambuf_type* s) noexcept;`

4 *Requires:* `s != nullptr` ~~shall not be a null pointer~~.

5 *Effects:* Initializes `sbuf_` with `s`.

24.8.4.2 ostreambuf_iterator operations [ostreambuf.iter.ops]

`ostreambuf_iterator`~~`<charT, traits>`~~`&`
`  operator=(charT c);`

1 *Requires:* `sbuf_ != `~~`0`~~`nullptr`.

2 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

3 *Returns:* `*this`.

`ostreambuf_iterator`~~`<charT, traits>`~~`& operator*();`

4 *Returns:* `*this`.

`ostreambuf_iterator`~~`<charT, traits>`~~`& operator++();`
`ostreambuf_iterator`~~`<charT, traits>&`~~` operator++(int);`

5 *Returns:* `*this`.

`bool failed() const noexcept;`

6 *Requires:* `sbuf_ != `~~`0`~~`nullptr`.

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

24.9 Range concepts [ranges]

24.9.1 General [ranges.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, predefined ranges, and stream ranges, as summarized in Table 5.

Table 5 — Ranges library summary

	Subclause	Header(s)
24.9.2	Requirements	
24.10	Range access	`<experimental/ranges/iterator>`
24.11	Range primitives	

24.9.2 Range requirements [ranges.requirements]

24.9.2.1 In general [ranges.requirements.general]

1 Ranges are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. It their simplest form, a range object is one on which one can call `begin` and `end` to get an iterator (24.2.6) and a sentinel (24.2.7). To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of ranges.

2 This document defines three fundamental categories of ranges based on the syntax and semantics supported by each: *range*, *sized range* and *view*, as shown in Table 6.

Table 6 — Relations among range categories

$$\begin{tikzcd} \textbf{Sized Range} \arrow[dr] & \\ & \textbf{Range} \\ \textbf{View} \arrow[ur] & \end{tikzcd}$$

3 The `Range` concept requires only that `begin` and `end` return an iterator and a sentinel. The `SizedRange` concept refines `Range` with the requirement that the number of elements in the range can be determined in constant time using the `size` function. The `View` concept specifies requirements on an `Range` type with constant-time copy and assign operations.

4 In addition to the three fundamental range categories, this document defines a number of convenience refinements of `Range` that group together requirements that appear often in the concepts, 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 type that satisfies `RandomAccessIterator` (24.2.13). The range categories *bidirectional ranges*, *forward ranges*, *input ranges*, and *output ranges* are defined similarly.

24.9.2.2 Ranges [ranges.range]

1 The `Range` concept defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and an `end` sentinel. [*Note:* Most algorithms requiring this concept simply forward to an `Iterator`-based algorithm by calling `begin` and `end`. — *end note*]

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

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

template <class T>
concept bool Range() {
  return requires(T&& t) {
    ranges::end(t);
```

```
  };
}
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `Range<T>()` is satisfied if and only if

(2.1) — `[begin(t),end(t))` denotes a range.

(2.2) — Both `begin(t)` and `end(t)` are amortized constant time and non-modifying. [*Note:* `begin(t)` and `end(t)` do not require implicit expression variants. — *end note*]

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

3 [*Note:* Equality preservation of both `begin` and `end` enables passing a `Range` whose iterator type satisfies `ForwardIterator` to multiple algorithms and making multiple passes over the range by repeated calls to `begin` and `end`. Since `begin` is not required to be equality preserving when the return type does not satisfy `ForwardIterator`, repeated calls might not return equal values or might not be well-defined; `begin` should be called at most once for such a range. — *end note*]

24.9.2.3 Sized ranges [ranges.sized]

1 The `SizedRange` concept specifies the requirements of a `Range` type that knows its size in constant time with the `size` function.

```
template <class>
constexpr bool disable_sized_range = false;

template <class T>
concept bool SizedRange() {
  return Range<T>() &&
    !disable_sized_range<remove_cv_t<remove_reference_t<T>>> &&
    requires(const remove_reference_t<T>& t) {
      { ranges::size(t) } -> ConvertibleTo<difference_type_t<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 `iterator_t<T>` satisfies `ForwardIterator`, `size(t)` is well-defined regardless of the evaluation of `begin(t)`. [*Note:* `size(t)` is otherwise not required be well-defined after evaluating `begin(t)`. For a `SizedRange` whose iterator type does not model `ForwardIterator`, for example, `size(t)` might only be well-defined if evaluated before the first call to `begin(t)`. — *end note*]

3 [*Note:* The `disable_sized_range` predicate provides a mechanism to enable use of range types with the library that meet the syntactic requirements but do not in fact satisfy `SizedRange`. A program that instantiates a library template that requires a `Range` with such a range type `R` is ill-formed with no diagnostic required unless `disable_sized_range<remove_cv_t<remove_reference_t<R>>>` evaluates to `true` (17.5.1.3). — *end note*]

24.9.2.4 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 `View`s are:

(2.1) — A `Range` type that wraps a pair of iterators.

(2.2) — A `Range` type that holds its elements by `shared_ptr` and shares ownership with all its copies.

(2.3) — A `Range` type that generates its elements on demand.

A container (23) is not a `View` since copying the container copies the elements, which cannot be done in constant time. — *end example*]

```
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>;
}
```

3 Since the difference between `Range` and `View` is largely semantic, the two are differentiated with the help of the `enable_view` trait. Users may specialize `enable_view` to derive from `true_type` or `false_type`.

4 For a type `T`, the value of `__view_predicate<T>` shall be:

(4.1) — If `enable_view<T>` has a member type `type`, `enable_view<T>::type::value`;

(4.2) — Otherwise, if `T` is derived from `view_base`, `true`;

(4.3) — Otherwise, if `T` is an instantiation of class template `initializer_list` (18.9), `set` (23.4.6), `multiset` (23.4.7), `unordered_set` (23.5.6), or `unordered_multiset` (23.5.7), `false`;

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

(4.5) — Otherwise, `true`.

24.9.2.5 Bounded ranges [ranges.bounded]

1 The `BoundedRange` concept specifies requirements of an `Range` type for which `begin` and `end` return objects of the same type. [*Note:* The standard containers (23) satisfy `BoundedRange`. — *end note*]

```
template <class T>
concept bool BoundedRange() {
  return Range<T>() && Same<iterator_t<T>, sentinel_t<T>>();
}
```

24.9.2.6 Input ranges [ranges.input]

1 The `InputRange` concept specifies requirements of an `Range` type for which `begin` returns a type that satisfies `InputIterator` (24.2.9).

```
template <class T>
concept bool InputRange() {
  return Range<T>() && InputIterator<iterator_t<T>>();
}
```

24.9.2.7 Output ranges [ranges.output]

1 The `OutputRange` concept specifies requirements of an `Range` type for which `begin` returns a type that satisfies `OutputIterator` (24.2.10).

```
template <class R, class T>
concept bool OutputRange() {
  return Range<R>() && OutputIterator<iterator_t<R>, T>();
}
```

24.9.2.8 Forward ranges [ranges.forward]

1 The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that satisfies `ForwardIterator` (24.2.11).

```
template <class T>
concept bool ForwardRange() {
  return InputRange<T>() && ForwardIterator<iterator_t<T>>();
}
```

24.9.2.9 Bidirectional ranges [ranges.bidirectional]

1 The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that satisfies `BidirectionalIterator` (24.2.12).

```
template <class T>
concept bool BidirectionalRange() {
  return ForwardRange<T>() && BidirectionalIterator<iterator_t<T>>();
}
```

24.9.2.10 Random access ranges [ranges.random.access]

1 The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that satisfies `RandomAccessIterator` (24.2.13).

```
template <class T>
concept bool RandomAccessRange() {
  return BidirectionalRange<T>() && RandomAccessIterator<iterator_t<T>>();
}
```

24.10 Range access [iterator.range]

1 In addition to being available via inclusion of the <iterator> header, the function templates in 24.10 are available when any of the following headers are included: <array>, <deque>, <forward_list>, <list>, <map>, <regex>, <set>, <string>, <unordered_map>, <unordered_set>, and <vector>.

```
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
```

2 *Returns:* `c.begin()`.

```
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
```

3 *Returns:* `c.end()`.

```
template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
```

4 *Returns:* `array`.

```
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
```

5 *Returns:* `array + N`.

```
template <class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
  -> decltype(std::begin(c));
```

6 *Returns:* `std::begin(c)`.

```
template <class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
  -> decltype(std::end(c));
```

7 *Returns:* `std::end(c)`.

```
template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());
```

8 *Returns:* `c.rbegin()`.

```
template <class C> auto rend(C& c) -> decltype(c.rend());
template <class C> auto rend(const C& c) -> decltype(c.rend());
```

9 *Returns:* `c.rend()`.

```
template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);
```

10 *Returns:* `reverse_iterator<T*>(array + N)`.

```
template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);
```

11 *Returns:* `reverse_iterator<T*>(array)`.

```
template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);
```

12 *Returns:* `reverse_iterator<const E*>(il.end())`.

```
template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);
```

13 *Returns:* `reverse_iterator<const E*>(il.begin())`.

```
template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));
```

14 *Returns:* `std::rbegin(c)`.

```
template <class C> auto crend(const C& c) -> decltype(std::rend(c));
```

15 *Returns:* `std::rend(c)`.

24.10.1 `begin` [iterator.range.begin]

1 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) — `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*]

(1.2) — Otherwise, `(E) + 0` if `E` has array type (3.9.2).

(1.3) — Otherwise, `DECAY_COPY((E).begin())` if its type `I` meets the syntactic requirements of `Iterator<I>()`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, `DECAY_COPY(begin(E))` if 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.

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

2 *Remark:* Whenever `ranges::begin(E)` is a valid expression, the type of `ranges::begin(E)` satisfies `Iterator`.

24.10.2 `end` [iterator.range.end]

1 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) — `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*]

(1.2) — Otherwise, `(E) + extent<T>::value` if `E` has array type (3.9.2) `T`.

(1.3) — Otherwise, `DECAY_COPY((E).end())` if its type `S` meets the syntactic requirements of `Sentinel<S, decltype(ranges::begin(E)>()`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, `DECAY_COPY(end(E))` if its type `S` meets the syntactic requirements of `Sentinel<S, decltype( ranges::begin(E))>()` with overload resolution performed in a context that includes the declaration `void end(auto&) = delete;` and does not include a declaration of `ranges::end`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

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

2 *Remark:* Whenever `ranges::end(E)` is a valid expression, the types of `ranges::end(E)` and `ranges::begin(E)` satisfy `Sentinel`.

24.10.3 `cbegin` [iterator.range.cbegin]

1 The name `cbegin` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::cbegin(E)` for some expression `E` of type `T` is equivalent to `ranges::begin((const T&)(E))`.

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

3 [*Note:* Whenever `ranges::cbegin(E)` is a valid expression, the type of `ranges::cbegin(E)` satisfies `Iterator`. — *end note*]

24.10.4 `cend` [iterator.range.cend]

1 The name `cend` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::cend(E)` for some expression `E` of type `T` is equivalent to `ranges::end((const T&)(E))`.

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

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

24.10.5 `rbegin` [iterator.range.rbegin]

1 The name `rbegin` denotes a customization point object (17.5.2.1.5). The effect of the expression `ranges::rbegin(E)` for some expression `E` is equivalent to:

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

(1.2) — Otherwise, `make_reverse_iterator((E) + extent<T>::value)` if `E` has array type (3.9.2) `T`.

(1.3) — Otherwise, `DECAY_COPY((E).rbegin())` if its type `I` meets the syntactic requirements of `Iterator<I>()`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, `make_reverse_iterator(ranges::end(E))` if both `ranges::begin(E)` and `ranges::end(E)` have the same type `I` which meets the syntactic requirements of `BidirectionalIterator<I>()` (24.2.12). If `BidirectionalIterator` is not satisfied, the program is ill-formed with no diagnostic required.

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

2 *Remark:* Whenever `ranges::rbegin(E)` is a valid expression, the type of `ranges::rbegin(E)` satisfies `Iterator`.

24.10.6 `rend` [iterator.range.rend]

1 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:

(1.1) — `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*]

(1.2) — Otherwise, `make_reverse_iterator((E) + 0)` if `E` has array type (3.9.2).

(1.3) — Otherwise, `DECAY_COPY((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.

(1.4) — 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>()` (24.2.12). If `BidirectionalIterator` is not satisfied, the program is ill-formed with no diagnostic required.

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

2 *Remark:* Whenever `ranges::rend(E)` is a valid expression, the types of `ranges::rend(E)` and `ranges::rbegin(E)` satisfy `Sentinel`.

24.10.7 `crbegin` [iterator.range.crbegin]

1 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))`.

2 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*]

3 [*Note:* Whenever `ranges::crbegin(E)` is a valid expression, the type of `ranges::crbegin(E)` satisfies `Iterator`. — *end note*]

24.10.8 `crend` [iterator.range.crend]

1 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))`.

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

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

24.11 Range primitives [range.primitives]

```
template <Range R>
difference_type_t<iterator_t<R>> distance(R&& r);
```

1 *Effects:* Equivalent to: `ranges::distance(ranges::begin(r), ranges::end(r))`

```
template <SizedRange R>
difference_type_t<iterator_t<R>> distance(R&& r);
```

2 *Effects:* Equivalent to: `ranges::size(r)`

24.11.1 `size` [range.primitives.size]

1 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:

(1.1) — `extent<T>::value` if `T` is an array type (3.9.2).

(1.2) — Otherwise, `DECAY_COPY(((const T&)(E)).size())` if its type `I` satisfies `Integral<I>()` and `disable_sized_range<T>` (24.9.2.3) is `false`.

(1.3) — Otherwise, `DECAY_COPY(size((const T&)(E)))` if its type `I` satisfies `Integral<I>()` 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`, and `disable_sized_range<T>` is `false`.

(1.4) — Otherwise, `DECAY_COPY(ranges::cend(E) - ranges::cbegin(E))`, except that `E` is only evaluated once, if the types `I` and `S` of `ranges::cbegin(E)` and `ranges::cend(E)` meet the syntactic requirements of `SizedSentinel<S, I>()` (24.2.8) and `ForwardIterator<I>()`. If `SizedSentinel` and `ForwardIterator` are not satisfied, the program is ill-formed with no diagnostic required.

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

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

24.11.2 `empty` [range.primitives.empty]

1 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:

(1.1) — `bool((E).empty())` if it is valid.

(1.2) — Otherwise, `ranges::size(E) != 0` if it is valid.

(1.3) — Otherwise, `bool(ranges::begin(E) != ranges::end(E))`, except that `E` is only evaluated once, if the type of `ranges::begin(E)` satisfies `ForwardIterator`.

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

2 *Remark:* Whenever `ranges::empty(E)` is a valid expression, it has type `bool`.

24.11.3 `data` [range.primitives.data]

1 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:

(1.1) — `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*]

(1.2) — Otherwise, `DECAY_COPY((E).data())` if it has pointer to object type.

(1.3) — Otherwise, `ranges::begin(E)` if it has pointer to object type.

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

2 *Remark:* Whenever `ranges::data(E)` is a valid expression, it has pointer to object type.

24.11.4 `cdata` [range.primitives.cdata]

1 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))`.

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

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

25 Algorithms library [algorithms]

25.1 General [algorithms.general]

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 23) and other sequences.

2 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

	Subclause	Header(s)
25.2	Non-modifying sequence operations	
25.3	Mutating sequence operations	`<experimental/ranges/algorithm>`
25.4	Sorting and related operations	
25.5	C library algorithms	`<cstdlib>`

Header `<experimental/ranges/algorithm>` synopsis

```
#include <initializer_list>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  namespace tag {
    // 25.1, tag specifiers (See 20.15.2):
    struct in;
    struct in1;
    struct in2;
    struct out;
    struct out1;
    struct out2;
    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<iterator_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<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<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<iterator_t<Rng>, Proj>> Fun>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    for_each(Rng&& rng, Fun f, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires 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<iterator_t<Rng>, Proj>, const T*>()
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Proj = identity,
    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{});
```

```
template <ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
    IndirectCallableRelation<iterator_t<Rng2>,
      projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});

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

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<projected<I, Proj>> Pred = equal_to<>>
  I
    adjacent_find(I first, S last, Pred pred = Pred{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectCallableRelation<projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng>
    adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});

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

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>()
  difference_type_t<iterator_t<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<projected<I, Proj>> Pred>
  difference_type_t<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});

// D.10 (deprecated):
```

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator 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, InputIterator 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{});

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<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>),
              tag::in2(safe_iterator_t<Rng2>)>
    mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// D.10 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator 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, InputIterator 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{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
  bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
```

```
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
  bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 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<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<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
  bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
  I1
    search(I1 first1, S1 last1, I2 first2, S2 last2,
           Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
  safe_iterator_t<Rng1>
    search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class T,
    class Pred = equal_to<>, class Proj = identity>
  requires IndirectlyComparable<I, const T*, Pred, Proj>()
  I
    search_n(I first, S last, difference_type_t<I> count,
```

```
              const T& value, Pred pred = Pred{},
              Proj proj = Proj{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>()
  safe_iterator_t<Rng>
    search_n(Rng&& rng, difference_type_t<iterator_t<Rng>> count,
             const T& value, Pred pred = Pred{}, Proj proj = Proj{});

// 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<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy(Rng&& rng, O result);

template <InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>()
  tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, difference_type_t<I> n, O result);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});

template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>()
  tagged_pair<tag::in(I1), tag::out(I2)>
    copy_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<Rng>, I>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    copy_backward(Rng&& rng, I result);

// 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<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    move(Rng&& rng, O result);

template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>()
  tagged_pair<tag::in(I1), tag::out(I2)>
    move_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<Rng>, I>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    move_backward(Rng&& rng, I result);

// 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<iterator_t<Rng>, I>()
  tagged_pair<tag::in1(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<iterator_t<Rng1>, iterator_t<Rng2>>()
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    swap_ranges(Rng1&& rng1, Rng2&& rng2);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>()
  tagged_pair<tag::in(I), tag::out(O)>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<R>, Proj>)>>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    transform(Rng&& rng, O result, F op, Proj proj = Proj{});

// D.10 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator 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{});

  // D.10 (deprecated):
  template <InputRange Rng, InputIterator 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{});

  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      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, 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>
    requires Writable<O, indirect_result_of_t<F&(
      projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>()
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
                 tag::in2(safe_iterator_t<Rng2>),
                 tag::out(O)>
      transform(Rng1&& rng1, Rng2&& rng2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

[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, class T2, class Proj = identity>
    requires Writable<I, const T2&>() &&
      IndirectCallableRelation<equal_to<>, projected<I, Proj>, const T1*>()
    I
      replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

  template <ForwardRange Rng, class T1, class T2, class Proj = identity>
    requires Writable<iterator_t<Rng>, const T2&>() &&
      IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>()
    safe_iterator_t<Rng>
      replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectCallablePredicate<projected<I, Proj>> Pred>
    requires Writable<I, const T&>()
    I
      replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

  template <ForwardRange Rng, class T, class Proj = identity,
      IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Writable<iterator_t<Rng>, const T&>()
```

```
  safe_iterator_t<Rng>
    replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O>() &&
    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, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
    IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
    class Proj = identity, IndirectCallablePredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>()
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

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

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

template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);

template <Callable F, OutputIterator<result_of_t<F&()>> O,
    Sentinel<O> S>
  O generate(O first, S last, F gen);

template <Callable F, OutputRange<result_of_t<F&()>> Rng>
  safe_iterator_t<Rng>
    generate(Rng&& rng, F gen);

template <Callable F, OutputIterator<result_of_t<F&()>> O>
  O generate_n(O first, difference_type_t<O> n, F gen);

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
```

```
    requires Permutable<I>() &&
      IndirectCallableRelation<equal_to<>, projected<I, Proj>, const T*>()
    I remove(I first, S last, const T& value, Proj proj = Proj{});

  template <ForwardRange Rng, class T, class Proj = identity>
    requires Permutable<iterator_t<Rng>>() &&
      IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>()
    safe_iterator_t<Rng>
      remove(Rng&& rng, const T& value, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectCallablePredicate<projected<I, Proj>> Pred>
    requires Permutable<I>()
    I remove_if(I first, S last, Pred pred, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
      IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
    requires Permutable<iterator_t<Rng>>()
    safe_iterator_t<Rng>
      remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
      class Proj = identity>
    requires IndirectlyCopyable<I, O>() &&
      IndirectCallableRelation<equal_to<>, projected<I, Proj>, const T*>()
    tagged_pair<tag::in(I), tag::out(O)>
      remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

  template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
      IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>()
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, 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<iterator_t<Rng>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<Rng>, O>()
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      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<iterator_t<Rng>, Proj>> R = equal_to<>>
    requires Permutable<iterator_t<Rng>>()
    safe_iterator_t<Rng>
```

```
      unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
      class Proj = identity, IndirectCallableRelation<projected<I, Proj>> R = equal_to<>>
    requires IndirectlyCopyable<I, O>() && (ForwardIterator<I>() ||
      ForwardIterator<O>() || IndirectlyCopyableStorable<I, O>())
    tagged_pair<tag::in(I), tag::out(O)>
      unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

  template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
      IndirectCallableRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
    requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
      (ForwardIterator<iterator_t<Rng>>() || ForwardIterator<O>() ||
       IndirectlyCopyableStorable<iterator_t<Rng>, O>())
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});

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

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

  template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>()
    tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);

  template <BidirectionalRange Rng, WeaklyIncrementable O>
    requires IndirectlyCopyable<iterator_t<Rng>, O>()
    tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
      reverse_copy(Rng&& rng, O result);
```

[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<iterator_t<Rng>>()
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
      rotate(Rng&& rng, iterator_t<Rng> middle);

  template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>()
    tagged_pair<tag::in(I), tag::out(O)>
      rotate_copy(I first, I middle, S last, O result);

  template <ForwardRange Rng, WeaklyIncrementable O>
```

```
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);

// 25.3.12, shuffle:
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I>() &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>>() &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>()
  I shuffle(I first, S last, Gen&& g);

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

// 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<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<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<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng>
    partition(Rng&& rng, Pred pred, Proj proj = Proj{});

template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng>
    stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity, 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<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O1>() &&
    IndirectlyCopyable<iterator_t<Rng>, O2>()
  tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
    partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  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<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<I1, I2>() && Sortable<I2, Comp, Proj2>() &&
```

```
      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<iterator_t<Rng1>, iterator_t<Rng2>>() &&
      Sortable<iterator_t<Rng2>, Comp, Proj2>() &&
      IndirectCallableStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>>()
  safe_iterator_t<Rng2>
    partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableStrictWeakOrder<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<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});

// 25.4.3, binary search:
template <ForwardIterator I, Sentinel<I> S, 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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

[Editor's note: This could return a `range` instead of a `pair`. See the Future Work annex (C.2).]

```
template <ForwardIterator I, Sentinel<I> S, 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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

// 25.4.4, merge:
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    merge(Rng1&& rng1, Rng2&& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
```

```
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});

// 25.4.5, set operations:
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 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  O
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  O
    set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
```

```
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  tagged_pair<tag::in1(I1), tag::out(O)>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 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<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
```

```
      class Proj = identity>
    requires Sortable<I, Comp, Proj>()
    I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>()
    safe_iterator_t<Rng>
      make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>()
    I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>()
    safe_iterator_t<Rng>
      sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
      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<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<projected<I, Proj>> Comp = less<>>
    I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <RandomAccessRange Rng, class Proj = identity,
      IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    safe_iterator_t<Rng>
      is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  // 25.4.7, minimum and maximum:
  template <class T, class Proj = identity,
      IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
    constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

[Editor's note: REVIEW: The Palo Alto report returns by const reference here but the current document returns by value.]

```
  template <Copyable T, class Proj = identity,
      IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
    constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    requires Copyable<value_type_t<iterator_t<Rng>>>()
    value_type_t<iterator_t<Rng>>
      min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <class T, class Proj = identity,
```

```
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>()
  value_type_t<iterator_t<Rng>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(T), tag::max(T)>
    minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>()
  tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
              tag::max(value_type_t<iterator_t<Rng>>)>
    minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    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,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
```

```
    tagged_pair<tag::min(safe_iterator_t<Rng>),
                tag::max(safe_iterator_t<Rng>)>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectCallableStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
    bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectCallableStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
    bool
      lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  // 25.4.9, permutations:
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>()
    bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>()
    bool
      next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>()
    bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>()
    bool
      prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}}}}
```

3 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.

4 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.]

5 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 (24.2.9). If an algorithm's template parameter

is `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the actual template argument shall satisfy the requirements of an output iterator (24.2.10). If an algorithm's template parameter is `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the actual template argument shall satisfy the requirements of a forward iterator (24.2.11). If an algorithm's template parameter is `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the actual template argument shall satisfy the requirements of a bidirectional iterator (24.2.12). If an algorithm's template parameter is `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the actual template argument shall satisfy the requirements of a random-access iterator (24.2.13).

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.[4] When such a version is provided for *algorithm* it is called *algorithm_copy*. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

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:* 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*]

11 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.

12 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

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

and that of `b-a` is the same as of

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

4) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

13 In the description of algorithm return values, sentinel values are sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator as follows:

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

14 Overloads of algorithms that take `Range` arguments (24.9.2.2) 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.

15 Some algorithms declare both an overload that takes a `Range` and an `Iterator`, and an overload that takes two `Range` parameters. Since an array type (3.9.2) both satisfies `Range` and decays to a pointer (4.2) which satisfies `Iterator`, such overloads are ambiguous when an array is passed as the second argument. Implementations shall provide a mechanism to resolve this ambiguity in favor of the overload that takes two ranges.

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

17 Despite that the algorithm declarations nominally accept parameters by value, it is unspecified when and if the argument expressions are used to initialize the actual parameters except that any such initialization shall be sequenced before (1.9) the algorithm returns. [*Note:* The behavior of a program that modifies the values of the actual argument expressions is consequently undefined unless the algorithm return happens before (1.10) any such modifications. — *end note*]

[Editor's note: The intent is to allow algorithm implementations to accept parameters by forwarding reference and delay or omit initialization of the actual parameter, either as an optimization or to implement the array disambiguation requirement described above.]

[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 [alg.tagspec]

```
namespace tag {
  struct in { /* implementation-defined */ };
  struct in1 { /* implementation-defined */ };
  struct in2 { /* implementation-defined */ };
  struct out { /* implementation-defined */ };
  struct out1 { /* implementation-defined */ };
  struct out2 { /* implementation-defined */ };
  struct fun { /* implementation-defined */ };
  struct min { /* implementation-defined */ };
  struct max { /* implementation-defined */ };
  struct begin { /* implementation-defined */ };
  struct end { /* implementation-defined */ };
}
```

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

2 `tag::`X is a tag specifier (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.

3 [*Example:* `tag::in` is a type such that *`TAGGET`*`(`D`, tag::in, `N`)` names a type with the following interface:

```
struct __input_getter {
```

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

— *end example*]

25.2 Non-modifying sequence operations [alg.nonmodifying]

25.2.1 All of [alg.all_of]

```
template <class InputIterator, class Predicate>
  bool all_of(InputIterator first, InputIterator last, Predicate pred);

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<iterator_t<Rng>, Proj>> Pred>
  bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if ~~`pred(*i)`~~`invoke(pred, invoke(proj, *i))` is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

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<projected<I, Proj>> Pred>
  bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `false` if `[first,last)` is empty or if there is no iterator `i` in the range `[first,last)` such that ~~`pred(*i)`~~`invoke(pred, invoke(proj, *i))` is `true`, and `true` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

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<projected<I, Proj>> Pred>
  bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if ~~`pred(*i)`~~`invoke(pred, invoke(proj, *i))` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

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<projected<iterator_t<Rng>, Proj>> Fun>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    for_each(Rng&& rng, Fun f, Proj proj = Proj{});
```

1 *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 `invoke(f, invoke(proj, *i))` for every iterator `i` in the range `[first,last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the ~~type of `first` satisfies the requirements of a mutable iterator~~result of `invoke(proj, *i)` is a mutable reference, `f` may apply nonconstant functions ~~through the dereferenced iterator~~. — *end note*]

3 *Returns:* ~~`std::move(f)`~~`{last, std::move(f)}`.

4 *Complexity:* Applies `f` and `proj` exactly `last - first` 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<iterator_t<Rng>, Proj>, const T*>()
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    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<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: ~~`*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{});

template <ForwardRange Rng1, ForwardRange Rng2,
    class Proj = identity,
    IndirectCallableRelation<iterator_t<Rng2>,
      projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following ~~corresponding~~ condition~~s~~ holds: ~~`*(i + n) == *(first2 + n)`, `pred(*(i + n), *(first2 + n)) != false`~~ `invoke(pred, invoke(proj, *(i + n)), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator

is found.

3 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate and projection.

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<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<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_first_of(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Finds an element that matches one of a set of values.

2 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following condition~~s~~ 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.

3 *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<I, 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<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng>
    adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding condition~~s~~ 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)` applications 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]

```
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*>()
  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<iterator_t<Rng>, Proj>, const T*>()
  difference_type_t<iterator_t<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<projected<I, Proj>> Pred>
  difference_type_t<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: ~~`*i == value, pred(*i) != false`~~`invoke(proj, *i) == value, invoke(pred, invoke(proj, *i)) != false`.

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

25.2.10 Mismatch [mismatch]

```
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);

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, InputIterator 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, InputIterator 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{});

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<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

2 *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~~invoke(pred, invoke(proj1, *i), invoke(proj2, *(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.

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

25.2.11 Equal [alg.equal]

```
template <class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

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, InputIterator 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, InputIterator 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{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
  bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
  bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Remarks:* If `last2` was not given in the argument list, it denotes `first2 + (last1 - first1)` below.

2 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if for every iterator `i` in the range `[first1,last1)` the following ~~corresponding~~ conditions holds: ~~`*i == *(first2 + (i - first1))`, `pred(*i, *(first2 + (i - first1))) != false`~~`invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1)))) != false`. Otherwise, returns `false`.

3 *Complexity:* No applications of the corresponding predicate and projections if ~~`InputIterator1` and `InputIterator2` meet the requirements of random access iterators~~ `SizedSentinel<S1, I1>()` is satisfied, and `SizedSentinel<S2, I2>()` 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 [alg.is_permutation]

```
template <class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
template <class ForwardIterator1, class ForwardIterator2,
                class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);
template <class ForwardIterator1, class ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2);
template <class ForwardIterator1, class ForwardIterator2,
                class BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      BinaryPredicate pred);

// 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<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<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
  bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
```

```
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *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 ~~`ForwardIterator2`~~`I2` `begin`, such that `equal(first1, last1, begin, pred, proj1, proj2)` returns `true` ~~or `equal(first1, last1, begin, pred)` returns `true`~~; otherwise, returns `false`.

4 *Complexity:* No applications of the corresponding predicate and projections if ~~`ForwardIterator1` and `ForwardIterator2` meet the requirements of random access iterators~~ `SizedSentinel<S1, I1>()` is satisfied, and `SizedSentinel<S2, I2>()` 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 `pred` was not given in the argument list or `equal(first1, last1, first2, last2, pred)` would return `true` if `pred` was given in the argument list~~; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `distance(first1, last1)`.

25.2.13 Search [alg.search]

```
template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);

template <class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
  I1
    search(I1 first1, S1 last1, I2 first2, S2 last2,
           Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
  safe_iterator_t<Rng1>
    search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following ~~corresponding~~ condition~~s~~ holds: ~~`*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`~~`invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) != false`. Returns `first1` if `[first2,last2)` is empty, otherwise returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and projections.

```
template <class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last, Size count,
             const T& value);

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, difference_type_t<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>()
  safe_iterator_t<Rng>
    search_n(Rng&& rng, difference_type_t<iterator_t<Rng>> count,
             const T& value, Pred pred = Pred{}, Proj proj = Proj{});
```

4 *Requires:* The type `Size` shall be convertible to integral type (4.7, 12.3).

5 *Effects:* Finds a subsequence of equal values in a sequence.

6 *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~~ condition~~s~~ holds: ~~`*(i + n) == value, pred(*(i + n),value) != false`~~`invoke(pred, invoke(proj, *(i + n)), value) != false`. Returns `last` if no such iterator is found.

7 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

25.3 Mutating sequence operations [alg.modifying.operations]

25.3.1 Copy [alg.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<tag::in(I), tag::out(O)>
    copy(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy(Rng&& rng, O result);
```

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

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

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

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

```
template <class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(InputIterator first, Size n,
                        OutputIterator result);
```

```
template <InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>()
  tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, difference_type_t<I> n, O result);
```

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, O result, Pred pred, Proj proj = Proj{});
```

```
template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

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

9 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which ~~`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::out(I2)>
    copy_backward(I1 first, S1 last, I2 result);
```

```
template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<Rng>, I>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    copy_backward(Rng&& rng, I result);
```

13 *Effects:* Copies elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to `first`.[5] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

14 *Requires:* `result` shall not be in the range `(first,last]`.

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

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

25.3.2 Move [alg.move]

```
template <class InputIterator, class OutputIterator>
  OutputIterator move(InputIterator first, InputIterator last,
                      OutputIterator result);
```

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>()
  tagged_pair<tag::in(I), tag::out(O)>
    move(I first, S last, O result);
```

```
template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyMovable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    move(Rng&& rng, O result);
```

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

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

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

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

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2
    move_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last,
                  BidirectionalIterator2 result);
```

```
template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>()
  tagged_pair<tag::in(I1), tag::out(I2)>
    move_backward(I1 first, S1 last, I2 result);
```

```
template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<Rng>, I>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    move_backward(Rng&& rng, I result);
```

5) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

5 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to first.[6] For each positive integer `n <= (last - first)`, performs `*(result - n) = std::move(*(last - n))`.

6 *Requires:* `result` shall not be in the range `(first,last]`.

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

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

25.3.3 swap [alg.swap]

```
template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2);
```

```
// 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<iterator_t<Rng>, I>()
  tagged_pair<tag::in1(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<iterator_t<Rng1>, iterator_t<Rng2>>()
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    swap_ranges(Rng1&& rng1, Rng2&& rng2);
```

1 *Effects:* For the first two overloads, let `last2` be `first2 + (last1 - first1)`. For each non-negative integer ~~`n < (last1 - first1)`~~ `n < min(last1 - first1, last2 - first2)` performs: `swap(*(first1 + n), *(first2 + n))`.

2 *Requires:* The two ranges `[first1,last1)` and ~~`[first2,first2 + (last1 - first1))`~~ `[first2,last2)` shall not overlap. `*(first1 + n)` shall be swappable with (19.2.10) `*(first2 + n)`.

3 *Returns:* ~~`first2 + (last1 - first1)`~~`{first1 + n, first2 + n}`, where `n` is `min(last1 - first1, last2 - first2)`.

4 *Complexity:* Exactly ~~`last1 - first1`~~`min(last1 - first1, last2 - first2)` swaps.

```
template <class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

5 *Effects:* `swap(*a, *b)`.

6 *Requires:* `a` and `b` shall be dereferenceable. `*a` shall be swappable with (19.2.10) `*b`.

6) `move_backward` should be used instead of move when last is in the range `[result - (last - first),result)`.

25.3.4 Transform [alg.transform]

```
template <class InputIterator, class OutputIterator,
         class UnaryOperation>
  OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op);

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>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>()
  tagged_pair<tag::in(I), tag::out(O)>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<R>, Proj>)>>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    transform(Rng&& rng, O result, F op, Proj proj = Proj{});

// D.10 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator 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{});

// D.10 (deprecated):
template <InputRange Rng, InputIterator 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{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    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, 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>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>()
```

```
tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
             tag::in2(safe_iterator_t<Rng2>),
             tag::out(O)>
  transform(Rng1&& rng1, Rng2&& rng2, O result,
            F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 For binary transforms that do not take `last2`, let `last2` be `first2 + (last1 - first1)`. Let N be `(last1 - first1)` for unary transforms, or `min(last1 - first1, last2 - first2)` for binary transforms.

2 *Effects:* Assigns through every iterator `i` in the range `[result,result +` ~~`(last1 - first1)`~~N`)` a new corresponding value equal to ~~`op(*(first1 + (i - result)))`~~`invoke(op, invoke(proj, *(first1 + (i - result))))` or ~~`binary_op(*(first1 + (i - result)), *(first2 + (i - result)))`~~`invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))`.

3 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,`~~`last1`~~`first1 +` N`]`, `[first2,first2 +` ~~`(last1 - first1)`~~N`]`, and `[result,result +` ~~`(last1 - first1)`~~N`]`.[7]

4 *Returns:* ~~`result + (last1 - first1)`~~`{first1 +` N`, result +` N`}` or `make_tagged_tuple<tag::in1, tag::in2, tag::out>(first1 +` N`, first2 +` N`, result +` N`)`.

5 *Complexity:* Exactly ~~`last1 - first1`~~N applications of `op` or `binary_op`.

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

25.3.5 Replace [alg.replace]

```
template <class ForwardIterator, class T>
  void replace(ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);

template <class ForwardIterator, class Predicate, class T>
  void replace_if(ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);

template <ForwardIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
  requires Writable<I, const T2&>() &&
    IndirectCallableRelation<equal_to<>, projected<I, Proj>, const T1*>()
  I
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <ForwardRange Rng, class T1, class T2, class Proj = identity>
  requires Writable<iterator_t<Rng>, const T2&>() &&
    IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>()
  safe_iterator_t<Rng>
    replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectCallablePredicate<projected<I, Proj>> Pred>
  requires Writable<I, const T&>()
  I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
```

7) The use of fully closed ranges is intentional.

```
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Writable<iterator_t<Rng>, const T&>()
  safe_iterator_t<Rng>
    replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
```

1 *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, class T2, OutputIterator<const 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, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
    IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
    class Proj = identity, IndirectCallablePredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>()
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});
```

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

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

template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);
```

1 ~~*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.

25.3.7 Generate [alg.generate]

```
template <class ForwardIterator, 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 <Callable F, OutputIterator<result_of_t<F&()>> O,
    Sentinel<O> S>
  O generate(O first, S last, F gen);
```

```
template <Callable F, OutputRange<result_of_t<F&()>> Rng>
  safe_iterator_t<Rng>
    generate(Rng&& rng, F gen);

template <Callable F, OutputIterator<result_of_t<F&()>> O>
  O generate_n(O first, difference_type_t<O> n, F gen);
```

1 *Effects:* ~~The first algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,last)`. The second algorithm 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.~~ Assigns the value of `invoke(gen)` through successive iterators in the range `[first,last)`, where `last` is `first + max(n, 0)` for `generate_n`.

2 *Requires:* `gen` takes no arguments, `Size` shall be convertible to an integral type (4.7, 12.3).

3 *Returns:* ~~`generate_n` returns `first + n` for non-negative values of `n` and `first` for negative values~~ `last`.

4 *Complexity:* Exactly `last - first`~~, `n`, or 0 invocations of `gen`~~ evaluations of `invoke(gen)` and assignments~~, respectively~~.

25.3.8 Remove [alg.remove]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I>() &&
    IndirectCallableRelation<equal_to<>, projected<I, Proj>, const T*>()
  I remove(I first, S last, const T& value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity>
  requires Permutable<iterator_t<Rng>>() &&
    IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>()
  safe_iterator_t<Rng>
    remove(Rng&& rng, const T& value, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<projected<I, Proj>> Pred>
  requires Permutable<I>()
  I remove_if(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectCallablePredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng>
    remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Requires:* The type of `*first` shall satisfy the `MoveAssignable` requirements (Table 22).

2 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: ~~`*i == value`~~`invoke(proj, *i) == value`, ~~`pred(*i) != false`~~`invoke(pred, invoke(proj, *i)) != false`.

3 *Returns:* The end of the resulting range.

4 *Remarks:* Stable (17.6.5.7).

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

6 *Note:* each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

```
template <class InputIterator, class OutputIterator, class T>
  OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);

template <class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);

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*>()
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
    IndirectCallableRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, 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<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

7 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap. ~~The expression `*result = *first` shall be valid.~~

8 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: ~~`*i == value`~~`invoke(proj, *i) == value`, ~~`pred(*i) != false`~~`invoke(pred, invoke(proj, *i)) != false`.

9 *Returns:* ~~T~~A pair consisting of `last` and the end of the resulting range.

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

11 *Remarks:* Stable (17.6.5.7).

25.3.9 Unique [alg.unique]

```
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<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng>
    unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
```

1 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: ~~`*(i - 1) == *i`~~`invoke(proj, *(i - 1)) == invoke(proj, *i)` or ~~`pred(*(i - 1), *i) != false`~~`invoke(pred, invoke(proj, *(i - 1)), invoke(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.

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

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallableRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
    (ForwardIterator<iterator_t<Rng>>() || ForwardIterator<O>() ||
     IndirectlyCopyableStorable<iterator_t<Rng>, O>())
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
```

5 *Requires:* ~~The comparison function shall be an equivalence relation.~~ The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.~~The expression `*result = *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). Otherwise `CopyConstructible` is not required.~~

6 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: ~~`*i == *(i`~~

~~- 1)~~invoke(proj, *i) == invoke(proj, *(i - 1)) or ~~pred(*i, *(i - 1)) != false~~invoke(pred, invoke(proj, *i), invoke(proj, *(i - 1))) != false.

7 *Returns:* ~~T~~A pair consisting of last and the end of the resulting range.

8 *Complexity:* For nonempty ranges, exactly last - first - 1 applications of the corresponding predicate and no more than twice as many applications of the projection.

25.3.10 Reverse [alg.reverse]

```
template <class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);

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

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

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

2 *Requires:* *first shall be swappable (19.2.10).

3 *Returns:* last.

4 *Complexity:* Exactly (last - first)/2 swaps.

```
template <class BidirectionalIterator, class OutputIterator>
  OutputIterator
    reverse_copy(BidirectionalIterator first,
                 BidirectionalIterator last, OutputIterator result);

template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>()
  tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);

template <BidirectionalRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    reverse_copy(Rng&& rng, O result);
```

5 *Effects:* Copies the range [first,last) to the range [result,result+(last-first)) such that for every non-negative integer i < (last - first) the following assignment takes place: *(result + (last - first) - 1 - i) = *(first + i).

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

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

8 *Complexity:* Exactly last - first assignments.

25.3.11 Rotate [alg.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<iterator_t<Rng>>()
  tagged_pair<tag::begin(safe_iterator_t<Rng>), tag::end(safe_iterator_t<Rng>)>
    rotate(Rng&& rng, iterator_t<Rng> middle);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* ~~`first + (last - middle)`~~`{first + (last - middle), last}`.

3 *Remarks:* This is a left rotate.

4 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges. ~~`ForwardIterator` shall satisfy the requirements of `ValueSwappable` (19.2.10). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and the requirements of `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<iterator_t<Rng>, O>()
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
```

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

7 *Returns:* ~~`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 [alg.random.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>() &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>>() &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>()
  I shuffle(I first, S last, Gen&& g);
```

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

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 *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 [alg.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<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~~iterators `i` for which `invoke(pred, invoke(proj, *i)) != false` come 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<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng>
    partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

4 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

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

5 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)` ~~`pred(*j) != false`~~`invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, ~~`pred(*k) == false`~~`invoke(pred, invoke(proj, *k)) == false`.

6 *Requires:* `ForwardIterator` shall satisfy the requirements of `ValueSwappable` (19.2.10).

7 *Complexity:* If ~~ForwardIterator~~I meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps are done; otherwise at most `last - first` swaps are done. Exactly `last - 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<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng>
    stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

8 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

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

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

10 *Requires:* `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).

11 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate and projection.

```
template <class InputIterator, class OutputIterator1,
          class OutputIterator2, class Predicate>
  pair<OutputIterator1, OutputIterator2>
  partition_copy(InputIterator first, InputIterator last,
                 OutputIterator1 out_true, OutputIterator2 out_false,
                 Predicate pred);
```

```
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<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O1>() &&
    IndirectlyCopyable<iterator_t<Rng>, O2>()
  tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
    partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
```

12 *Requires:* ~~InputIterator's value type shall be CopyAssignable, and shall be writable to the out_true and out_false OutputIterators, 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)`~~`invoke(pred, invoke(proj, *i))` is `true`, or to the output range beginning with `out_false` otherwise.

14 *Returns:* A ~~pair~~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 <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<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
```

16 *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.

17 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

18 *Complexity:* $\mathcal{O}(log(last - first))$ applications of `pred` and `proj`.

25.4 Sorting and related operations [alg.sorting]

1 All the operations in 25.4 ~~have two versions: one that takes a function object of type Compare and one that uses an operator<~~take an optional binary callable predicate of type `Comp` that defaults to `less<>`.

2 ~~Compare~~`Comp` is ~~a function object type (20.9)~~a callable object (20.9.2). The return value of ~~the function call operation~~the `invoke` operation 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.

3 ~~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).]

4 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:

(4.1) — comp(a, b) && comp(b, c) implies comp(a, c)

(4.2) — equiv(a, b) && equiv(b, c) implies equiv(a, c) [*Note:* Under these conditions, it can be shown that

(4.2.1) — equiv is an equivalence relation

(4.2.2) — comp induces a well-defined relation on the equivalence classes determined by equiv

(4.2.3) — The induced relation is a strict total ordering. — *end note*]

5 A sequence is *sorted with respect to a comparator* *and projection* comp and proj if for every iterator i pointing to the sequence and every non-negative integer n such that i + n is a valid iterator pointing to an element of the sequence, ~~comp(*(i + n), *i) == false~~invoke(comp, invoke(proj, *(i + n)), invoke(proj, *i)) == false.

6 A sequence [start,finish) is *partitioned with respect to an expression* f(e) if there exists an integer n such that for all 0 <= i < distance(start, finish), f(*(start + i)) is true if and only if i < n.

7 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an operator==, but an equivalence relation induced by the strict weak ordering. That is, two elements a and b are considered equivalent if and only if !(a < b) && !(b < a).

25.4.1 Sorting [alg.sort]

25.4.1.1 sort [sort]

```
template <class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
```

```
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *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).

3 *Complexity:* $\mathcal{O}(N\log(N))$ (where N `== last - first`) comparisons.

25.4.1.2 `stable_sort` [stable.sort]

```
template <class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *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).

3 *Complexity:* It does at most $N\log^2(N)$ (where N `== last - first`) comparisons; if enough extra memory is available, it is $N\log(N)$.

4 *Remarks:* Stable (17.6.5.7).

25.4.1.3 `partial_sort` [partial.sort]

```
template <class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});
```

1 *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

2 *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).

3 *Complexity:* It takes approximately `(last - first) * log(middle - first)` comparisons.

25.4.1.4 `partial_sort_copy` [partial.sort.copy]

```
template <class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);

template <class InputIterator, class RandomAccessIterator,
          class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);

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<iterator_t<Rng1>, iterator_t<Rng2>>() &&
      Sortable<iterator_t<Rng2>, Comp, Proj2>() &&
      IndirectCallableStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>>()
  safe_iterator_t<Rng2>
    partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

2 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

3 *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 `MoveAssignable` (Table 22).

4 *Complexity:* Approximately `(last - first) * log(min(last - first, result_last - result_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<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<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* `is_sorted_until(first, last, comp, proj) == last`

```
template <class ForwardIterator, class Compare>
  bool is_sorted(ForwardIterator first, ForwardIterator last,
    Compare comp);
```

2 *Returns:* `is_sorted_until(first, last, comp) == last`

```
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 comp);

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<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

3 *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.

4 *Complexity:* Linear.

25.4.2 Nth element [alg.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<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});
```

1 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first,nth)` and every iterator `j` in the range `[nth,last)` it holds that: ~~`!(*j < *i)` or `comp(*j, *i) == false`~~ `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == 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).~~

3 *Complexity:* Linear on average.

25.4.3 Binary search [alg.binary.search]

1 All of the algorithms in this section are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the ~~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, 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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression ~~`e < value`~~`invoke(comp, invoke(proj, e), value)` ~~or `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 condition~~s~~ holds: ~~`*j < value` or `comp(*j, value) != false`~~`invoke(comp, invoke(proj, *j), value) != false`.

3 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ ~~comparisons~~applications of the comparison function 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, 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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression ~~`!(value < e)`~~`!invoke(comp, value, invoke(proj, e))` ~~`or !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 condition~~s~~ holds: ~~`!(value < *j)` or `comp(value, *j) == false`~~`invoke(comp, value, invoke(proj, *j)) == false`.

3 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ ~~comparisons~~applications of the comparison function and projection.

25.4.3.3 `equal_range` [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, 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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions ~~`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:*

```
make_pair(lower_bound(first, last, value),
          upper_bound(first, last, value))
```

or

~~`make_pair(`~~`{lower_bound(first, last, value, comp, proj),`
`upper_bound(first, last, value, comp, proj)`~~`)`~~`}`

3 *Complexity:* At most $2 * \log_2(\texttt{last - first}) + \mathcal{O}(1)$ ~~comparisons~~applications of the comparison function and projection.

25.4.3.4 `binary_search` [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);

template <ForwardIterator I, Sentinel<I> S, 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, class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions ~~`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` ~~or `comp(*i, value) == false && comp(value, *i) == false`~~.

3 *Complexity:* At most $\log_2(\texttt{last - first}) + \mathcal{O}(1)$ ~~comparisons~~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,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    merge(Rng1&& rng1, Rng2&& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`~~, such that the resulting range satisfies `is_sorted(result, result_last)` or `is_sorted(result, result_last, comp)`, respectively~~. If an element `a` precedes `b` in an input range, `a` is copied into the output range before `b`. If `e1` is an element of `[first1,last1)` and `e2` of `[first2,last2)`, `e2` is copied into the output range before `e1` if and only if `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` is `true`.

2 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to ~~`operator<` or~~ `comp`, `proj1`, and `proj2`. The resulting range shall not overlap with either of the original ranges.

3 *Returns:* ~~`result + (last1 - first1) + (last2 - first2)`~~ `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result_last)`.

4 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` ~~comparisons~~applications of the comparison function and each projection.

5 *Remarks:* Stable (17.6.5.7).

```
template <class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
```

```
  requires Sortable<I, Comp, Proj>()
  I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});
```

6 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition ~~`*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))`~~ `invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))` will be false.

7 *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to ~~`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).~~

8 *Returns:* `last`

9 *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.

10 *Remarks:* Stable (17.6.5.7).

25.4.5 Set operations on sorted structures [alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They also work with `multiset`s (23.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multiset`s 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` [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 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
```

```
includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
         Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` ~~comparisons~~applications of the comparison function and projections.

25.4.5.2 `set_union` [set.union]

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);

template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_union(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_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* ~~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` [set.intersection]

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_intersection(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>()
  O
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  O
    set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` ~~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, 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` [set.difference]

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);

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, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* ~~The end of the constructed range~~`{last1, 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, the last $\max(m-n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

25.4.5.5 `set_symmetric_difference` [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 O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>()
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
```

```
    set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* ~~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 $|m - n|$ of those elements shall be copied to the output range: the last $m - n$ of these elements from `[first1,last1)` if $m > n$, and the last $n - m$ of these elements from `[first2,last2)` if $m < n$.

25.4.6 Heap operations [alg.heap.operations]

1 A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key properties are:

(1) There is no element greater than `*a` in the range and

(2) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $\mathcal{O}(\log(N))$ time.

2 These properties make heaps useful as priority queues.

3 `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

25.4.6.1 push_heap [push.heap]

```
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, Proj>()
  I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

2 *Requires:* The range `[first,last - 1)` shall be a valid heap. ~~The type of `*first` shall satisfy the `MoveConstructible` requirements (Table 20) and the `MoveAssignable` requirements (Table 22)~~.

3 *Returns:* `last`

4 *Complexity:* At most `log(last - first)` ~~comparisons~~applications of the comparison function and projection.

25.4.6.2 pop_heap [pop.heap]

```
template <class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                Compare comp);

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The range `[first,last)` shall be a valid non-empty heap. ~~`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).~~

2 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

3 *Returns:* `last`

4 *Complexity:* At most `2 * log(last - first)` ~~comparisons~~applications of the comparison function and projection.

25.4.6.3 make_heap [make.heap]

```
template <class RandomAccessIterator>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>()
  I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Constructs a heap out of the range `[first,last)`.

2 *Requires:* The type of `*first` shall satisfy the `MoveConstructible` requirements (Table 20) and the `MoveAssignable` requirements (Table 22).

3 *Returns:* `last`

4 *Complexity:* At most `3 * (last - first)` ~~comparisons~~applications of the comparison function and projection.

25.4.6.4 `sort_heap` [sort.heap]

```
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>()
  I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>()
  safe_iterator_t<Rng>
    sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts elements in the heap `[first,last)`.

2 *Requires:* The range `[first,last)` shall be a valid heap. ~~`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).~~

3 *Returns:* `last`

4 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`).

25.4.6.5 `is_heap` [is.heap]

```
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<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* `is_heap_until(first, last, comp, proj) == last`

```
template <class RandomAccessIterator, class Compare>
  bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

2 *Returns:* `is_heap_until(first, last, comp) == last`

```
template <class RandomAccessIterator>
  RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template <class RandomAccessIterator, class Compare>
  RandomAccessIterator is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <RandomAccessRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

3 *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.

4 *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 <class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  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 <Copyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T min(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>()
  value_type_t<iterator_t<Rng>>
    min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

4 *Requires:* ~~T is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`~~`distance(rng) > 0`.

5 *Returns:* The smallest value in the initializer_list or range.

6 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

```
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 <class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

7 *Requires:* Type `T` is `LessThanComparable` (Table 18).

8 *Returns:* The larger value.

9 *Remarks:* Returns the first argument when the arguments are equivalent.

```
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 <Copyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T max(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>()
  value_type_t<iterator_t<Rng>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

10 *Requires:* ~~T is LessThanComparable and CopyConstructible and t.size() > 0~~distance(rng) > 0.

11 *Returns:* The largest value in the initializer_list or range.

12 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

```
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 <class T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

13 *Requires:* Type T shall be LessThanComparable (Table 18).

14 *Returns:* ~~pair<const T&, const T&>(~~{b, a~~)~~} if b is smaller than a, and ~~pair<const T&, const T&>(~~{a, b~~)~~} otherwise.

15 *Remarks:* Returns ~~pair<const T&, const T&>(~~{a, b~~)~~} when the arguments are equivalent.

16 *Complexity:* Exactly one comparison and exactly two applications of the projection.

```
template <class T>
  constexpr pair<T, T> minmax(initializer_list<T> t);
template <class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

```
template <Copyable T, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(T), tag::max(T)>
    minmax(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <InputRange Rng, class Proj = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng>, Proj> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>()
  tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
              tag::max(value_type_t<iterator_t<Rng>>)>
    minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

17 *Requires:* ~~T is LessThanComparable and CopyConstructible and t.size() > 0~~distance(rng) > 0.

18 *Returns:* ~~pair<T, T>(~~{x, y~~)~~}, where x has the smallest and y has the largest value in the initializer list or range.

19 *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.

20 *Complexity:* At most ~~`(3/2) * t.size()`~~`(3/2) * distance(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<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<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

21 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition~~s~~ holds: ~~`!(*j < *i)` or `comp(*j, *i) == false`~~ `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`. Returns `last` if `first == last`.

22 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the ~~corresponding comparisons~~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<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<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

23 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition~~s~~ holds: ~~`!(*i < *j)` or `comp(*i, *j) == false`~~`invoke(comp, invoke(proj, *i), invoke(proj, *j)) == false`. Returns `last` if `first == last`.

24 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the ~~corresponding comparisons~~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<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<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::min(safe_iterator_t<Rng>),
              tag::max(safe_iterator_t<Rng>)>
    minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

25 *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.

26 *Complexity:* At most $max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the ~~corresponding predicate~~comparison function and at most twice as many applications of the projection, where N is distance(first, last).

25.4.8 Lexicographical comparison [alg.lex.comparison]

```
template <class InputIterator1, class InputIterator2>
  bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
  bool
    lexicographical_compare(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
    lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallableStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* true if the sequence of elements defined by the range [first1,last1) is lexicographically less than the sequence of elements defined by the range [first2,last2) and false otherwise.

2 *Complexity:* At most 2*min((last1 - first1), (last2 - first2)) applications of the corresponding comparison and projection.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise,

the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++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;
```

4 *Remarks:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

25.4.9 Permutation generators [alg.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<iterator_t<Rng>, Comp, Proj>()
  bool
    next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to ~~`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`.

2 *Requires:* `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (19.2.10).

3 *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<iterator_t<Rng>, Comp, Proj>()
  bool
    prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

4 *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`.

5 *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`.

6 *Requires:* `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (19.2.10).

7 *Complexity:* At most `(last - first)/2` swaps.

25.5 C library algorithms [alg.c.library]

1 Table 8 describes some of the contents of the header `<cstdlib>`.

Table 8 — Header `<cstdlib>` synopsis

Type	Name(s)	
Type:	`size_t`	
Functions:	`bsearch`	`qsort`

2 The contents are the same as the Standard C library header `<stdlib.h>` with the following exceptions:

3 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.

4 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 [numerics]

Header <experimental/ranges/random> synopsis

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class G>
  concept bool UniformRandomNumberGenerator() { return see below; }
}}}}
```

26.5 Random number generation [rand]

26.5.1 Requirements [rand.req]

26.5.1.3 Uniform random number generator requirements [rand.req.urng]

```
// defined in <experimental/ranges/random>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class G>
  concept bool UniformRandomNumberGenerator() {
    return requires(G g) {
      { g() } -> UnsignedIntegral; // not required to be equality preserving
      { G::min() } -> Same<result_of_t<G&()>>;
      { G::max() } -> Same<result_of_t<G&()>>;
    };
  }
}}}}
```

1 A *uniform random number generator* `g` of type `G` is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note:* The degree to which `g`'s results approximate the ideal is often determined statistically. — *end note*]

[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 `UniformRandomNumberGenerator<G>()` is satisfied if and only if

(2.1) — Both `G::min()` and `G::max()` are constant expressions (5.19).

(2.2) — `G::min() < G::max()`.

(2.3) — `G::min() <= g()`.

(2.4) — `g() <= G::max()`.

(2.5) — `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, InputIterator 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, InputIterator 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, InputIterator 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, InputIterator 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, InputIterator 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, InputIterator 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 [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.

The authors 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 [3] 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.

We would also like to thank the members of the Ranges SG who offered feedback on early drafts, especially Tony Van Eerd and Walter Brown.

Christopher Di Bella has contributed many editorial fixes.

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 constrained 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 verbose 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 `EqualityComparable` and the latter with `StrictTotallyOrdered`). Similar constraints are added to the other function objects in `<functional>`. 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-compatibliity, 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 alias template. This is to 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 `InputIterator`.

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 described in N4128 (1.2). This causes a number of differences 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 $\mathscr{O}(N)$ 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 compexity 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]

1 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]

1 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 $\mathscr{O}(N)$. 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 `SizedSentinel` (24.2.8). The `SizedSentinel` 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]

1 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. Even if not strictly needed, its absence would be surprising.

B.2.6 No Algorithm Reformulations [diff.n3351.no_eop_algos]

1 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 [5]). 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 [future]

1 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 [future.proxy]

1 As early at 1998 when Herb Sutter published his "When is a Container not a Container" [6] 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.

2 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 [4].) 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 [future.iterator_range]

1 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.

2 A future paper will propose such a type.

C.3 Range Views and Actions [future.views]

1 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.

2 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 [future.facade]

1 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.

2 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 [future.infrng]

1 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.

2 The problem with infinite ranges is the requirement that `WeaklyIncrementable` types have an associated `difference_type_t`. The `difference_type_t` 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 `difference_type_t`.

3 Given the fact that there are no infinite precision built-in integral types, the presence of `difference_type_t` in the concept requirements places a hard upper limit on how big ranges can be. This is a severe limitation.

4 Some observations:

(4.1) — Not all possible ranges have finitely countable elements.

(4.2) — Even a range with finitely countable elements may not be countable within the precision of the built-in integral types.

(4.3) — Not all algorithms require the ability to count increments. Algorithms like `distance` and `count` require finitely countable iterators, but `find` and `accumulate` do not.

5 The above observations suggest that there is a factorization of the existing concept hierarchy that could solve the problem. Some `Incrementable`s are finitely countable with a built-in integral `difference_type_t`, and some are not. The algorithms that require countability must say so with an extra requirement.

6 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_iterator`s. 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 [future.commontype]

1 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 [future.numcont]

1 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 Verbosity in Algorithm Constraints [future.constraints]

1 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 [future.initlsts]

1 Algorithms that do not mutate their input should accept `initializer_list` arguments wherever `Iterable`s are allowed. This requires extra overloads in places.

Annex E (informative) Reference implementation for `tagged` [tagged]

1 Below is a reference implementation of the `tagged` class template described in 20.15.2, and also `tagged_pair` (20.15.4), `tagged_tuple` (20.15.5), and `tag::in` (25.1).

```
namespace std { namespace experimental { namespace ranges { 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, TagSpecifier... Tags>
    requires sizeof...(Tags) >= tuple_size<Base>::value
  struct tagged;
  }}}

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

  namespace experimental { namespace ranges { inline namespace v1 {

  struct __getters {
  private:
    template <class, TagSpecifier...>
      requires sizeof...(Tags) >= tuple_size<Base>::value
```

```
    friend struct tagged;
    template <class Type, class Indices, TagSpecifier... Tags>
    struct collect_;
    template <class Type, std::size_t... Is, 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, TagSpecifier... Tags>
        requires sizeof...(Tags) <= tuple_size<Base>::value
      friend struct tagged;
      ~collect_() = default;
    };
    template <class Type, TagSpecifier... Tags>
    using collect = collect_<Type, make_index_sequence<sizeof...(Tags)>, Tags...>;
  };

  template <class Base, TagSpecifier... Tags>
  struct tagged
    : Base, __getters::collect<tagged<Base, Tags...>, Tags...> {
    using Base::Base;
    tagged() = default;
    tagged(tagged&&) = default;
    tagged(const 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>() && !Same<decay_t<U>, tagged>()
    tagged& operator=(U&& u) {
      static_cast<Base&>(*this) = std::forward<U>(u);
      return *this;
    }
  };
```

```
  template <TaggedType F, 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 <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 __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 __getters;
        ~getter() = default;
      };
    }
    // Other tag specifiers defined similarly, see 25.1
  }
}}}}
```

Annex F (informative) Iterator Design Rationale [sentrework]

1 The design of the iterator and sentinel concepts in this document differs substantially from the design presented in N4560. In short, N4560 had separate "weak" and "strong" variants of `Iterator`, `InputIterator`, and `OutputIterator`; in the current design the "strong" (`EqualityComparable` (19.3.3)) variants have been removed and the `Weak` prefix has been elided from the names of the "weak" variants. This rationale justifies the change and clarifies that it does not diminish the expressive power of iterator / sentinel pairs and ranges.

F.1 Problem statement [sentrework.problem]

1 Consider a threshold sentinel `S` with state consisting of a single integer `bound`. When compared for equality with an iterator, this sentinel's `==` returns `true` if and only if the iterator denotes a value greater than or equal to `bound`. `S` seems like a useful tool, and it's simple enough to be described completely in a couple of sentences. One would expect it to be expressible within the sentinel model of N4560. Given an array of integers:

```
int i[] = {2,1,3};
```

By any sensible definition of "the same range," `[i+1,S{2})` and `[i+1,S{3})` (where `S{n}` is a threshold sentinel with `bound` of `n`) denote the same range. `*(i+1) == 1` is less than both bounds (`2` and `3`), and `*(i+2) == 3` is greater than or equal to both bounds, so both ranges have length of one and denote only the single integer with value `1`.

2 N4560 requires iterators and their sentinels to satisfy the cross-type `EqualityComparable<T,U>()` concept (19.3.3). `EqualityComparable` effectively requires the values of the two participant types `T` and `U` to be put into correspondence: if two objects `t1` and `t2` of one type both compare equal to some value of the other type, those objects must be equal. The meaning of "be equal" is very clearly defined in the Ranges TS by the semantics of equality preserving expressions (19.1.1). If `t1` and `t2` are equal, the result of evaluating any expression required to be equality preserving must be the same for both `t1` and `t2`. Informally, the correspondence enforced by `EqualityComparable` requires values of the participant types to both represent the same abstract entities as personified by their common type.

3 Returning to the example above, the consequences of the `EqualityComparable` requirement are clear: since `i+2 == S{2}` and `i+2 == S{3}`, `S{2}` and `S{3}` must be equal. If they are equal, then the ranges `[i+0, S{2})` and `[i+0,S{3})` must also denote the same range due to the equality preservation of the expressions involved. This is, however, not the case since `*(i+0) == 2` is greater than or equal to `2` but not greater than or equal to `3`. `EqualityComparable<S,int*>()` and therefore `Sentinel<S,int*>()` are not satisfied; `S` is not a valid sentinel for `int*`.

4 Clearly there is a large class of useful sentinels that the N4560 model does not admit: stateful sentinels which might compare equal to the same iterator when they have differing state. N4560 constrains iterators and sentinels to both represent the abstract notion of "position," which is not a good fit for stateful sentinels.

F.2 Fixing the problem [sentrework.approach]

1 It seems odd to claim that the sentinel semantics are broken when there are two working implementations of the TS ([3], [2]) that haven't had any issues due to this semantic inconsistency. The clear implication is that the semantics of the N4560 model are not an exact match for the requirements of the algorithms. The semantics are likely overconstrained and could be relaxed without losing the properties the algorithms need to support equational reasoning.

F.2.1 Look at the Algorithms [sentrework.approach.alg]

1 As is often the case, a problem in the semantics is best fixed by examining the algorithms and determining what behaviors they require to ensure their postconditions. The properties the algorithms require for sentinels are (where `s` denotes a sentinel value and `i` an iterator value):

(1.1) — `i == s`, `s == i`, `i != s`, and `s != i` are equality-preserving expressions with the same domain.

(1.2) — Symmetry: `bool(i == s) == bool(s == i)` and `bool(i != s) == bool(s != i)`.

(1.3) — Complement: `bool(i != s) == !bool(i == s)`.

(1.4) — `i == s` is well-defined when `[i,s)` denotes a range.

2 The first three properties aren't specific to iterators and sentinels; they are general characteristics of the operators `==` and `!=` that are intuitively expected to hold whenever they are defined. As such, the `EqualityComparable` concepts should require them to hold and this particular set of properties can then be presented as a relaxation of `EqualityComparable<T,U>()`. The current design does exactly that, defining the concept `WeaklyEqualityComparable` (19.3.3) with exactly the specified semantic requirements, and reformulating both the single and cross-type variants of `EqualityComparable` as refinements thereof.

3 Notably, nothing in this list of properties requires comparison of sentinels with other sentinels – it's not a useful feature for generic code. The requirement for equality comparison of sentinels in N4560 is a checkbox feature to enable satisfaction of cross-type `EqualityComparable`. Absent the `EqualityComparable` requirement, nothing in the design requires equality comparison for sentinels. It's nonsensical to require sentinels to have `Regular` types if `==` is never required to be valid or useful; they should be `Semiregular`.

4 The list of properties also does not require that iterator types be `EqualityComparable` to participate in the `Sentinel` relationship. Just as removing the cross-type `EqualityComparable` requirement makes it unnecessary for sentinels to be `EqualityComparable` with other sentinels, it also removes the necessity for the iterator type to be `EqualityComparable`. In N4560 terms, `Sentinel` should be a relationship between a `WeakIterator` type that denotes an element of a range and a `Semiregular` type that represents an end-of-range calculation as a predicate over `WeakIterator` values embodied in the form of the `==` operator, i.e.:

```
template <class S, class I>
concept bool Sentinel() {
  return Semiregular<S>() && WeakIterator<I>() &&
    WeaklyEqualityComparable<S, I>();
}
```

a non-weak iterator is capable of both denoting an element and the end of a range. In other words, a "strong" iterator is a type that is simultaneously a weak iterator and a sentinel for that weak iterator:

```
template <class I>
concept bool Iterator() {
  return Sentinel<I, I>() &&
    EqualityComparable<I>(); // Keep the semantic property that == means
                             // substitutable in equality-preserving expressions
}
```

F.3 Ramifications [sentrework.ram]

1 Correcting the inconsistency in the `Sentinel` semantics creates a class of "weak" ranges – ranges with `WeakInput` or `WeakOutput` iterator types – that the algorithms do not admit. It would appear that algorithms that accept input (output) ranges should possibly be relaxed to accept weak input (weak output) ranges instead. How would doing so constrain the implementations of the algorithms? Do weak ranges have less expressive power?

F.3.1 Single-pass Algorithms [sentrework.ram.singlepass]

1 Algorithms that accept input or output iterators necessarily perform at most one pass over the range of elements those iterators reference. N4560's weak input and weak output iterators are consistent with those single-pass semantics. The sole distinction is support for equality comparison. How do algorithms use equality comparison with iterators for single-pass ranges?

2 Oddly enough, the answer to the question posed is that algorithms need not ever use equality comparison on single-pass iterators. In the iterator pair model its necessary to compare an iterator denoting the current position in the range with the supplied end iterator to determine when the algorithm reaches the end of the range. An algorithm *could* compare the "current" iterator with itself or a copy thereof, but it would be pointless to do so. An algorithm *cannot* compare the "current" iterator with a copy of an earlier value it held because that operation is not required to be valid for single-pass iterators.

3 The case is similar for the iterator / sentinel model. An algorithm compares the iterator with the sentinel to determine when the end of the range is reached, but cannot compare the iterator with a copy of an older value. It would again be pointless to compare the iterator with itself or a copy; there's no reason to test things known to be equal for equality. Consequently, relaxing the algorithms that use single-pass ranges to "weak" single-pass ranges doesn't reduce their power.

F.3.2 Why "strong" Iterators? [sentrework.ram.merge]

1 Relaxing the algorithms necessitates relaxing the iterator operations (`advance` et al. (24.6.5)), which the algorithms use. Indeed there are many components in the specification that require a "strong" iterator not because they need to compare iterator values for equality, but because iterator / sentinel pairs in N4560 necessarily used strong iterators. Once all such occurrences are relaxed, it becomes apparent that there is no need for separate "weak" and "strong" concepts for input and output iterators; the strong concepts are unused. The current design removes N4560's `InputIterator` and `OutputIterator` concepts and strips the `Weak` prefix from the names of N4560's `WeakInputIterator` and `WeakOutputIterator` since the name distinction is no longer necessary.

2 After removing the distinction between weak and strong input and output iterators, the `Iterator` concept is insufficiently distinguished from `Sentinel` to deserve a separate named concept. The current design pushes `Iterator`'s `Sentinel<I,I>()` requirement up into `ForwardIterator` so that the semantics of forward and stronger iterators are unchanged, and pushes the `EqualityComparable` requirement from `Iterator` down into `Sentinel`, but only requires it when the iterator and sentinel types are the same. `Iterator` may then be struck from the design, and the `Weak` prefix stripped from `WeakIterator` as well.

3 The end result of the design changes is to simplify the iterator concepts overall, resulting in a reduction of the number of iterator categories from N4560's seven to five as in C++14.

F.4 Conclusions [sentrework.conc]

1 The Ranges TS originally inherited the notions of distinct "weak" and "strong" kinds of iterators from the Palo Alto report (1.2). The distinction was necessary in the Palo Alto design to capture the difference between a lone iterator that need not support equality comparison and iterator pairs that must support equality comparison to be capable of denoting ranges. The addition of sentinels to the TS design makes that distinction unnecessary: iterators denote elements, ranges are denoted by iterator / sentinel pairs. A homogeneous pair of iterators can denote a range by satisfying the `Sentinel` concept, there's no need for separate "weak" and "strong" iterator concepts muddying the waters.

Bibliography

[1] Adobe source libraries. http://stlab.adobe.com. Accessed: 2014-10-8.

[2] Casey Carter and Eric Niebler. CMCSTL2: An implementation of C++ extensions for ranges. https://github.com/caseycarter/cmcstl2/. Accessed: 2015-12-07.

[3] Eric Niebler. Range-v3. https://github.com/ericniebler/range-v3. Accessed: 2015-4-6.

[4] Eric Niebler. To be or not to be (an iterator). http://ericniebler.com/2015/01/28/to-be-or-not-to-be-an-iterator/. Accessed: 2015-4-6.

[5] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[6] Herb Sutter. When is a container not a container? *C++ Report*, 14, 5 1999. http://www.gotw.ca/publications/mill09.htm.

[7] Jeffrey Yasskin. N3350: A minimal std::range<iter>, 1 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3350.html.

Index

Index of library names