Document Number: N4382
Date: 2015-04-12
Reply to: Eric Niebler
eric.niebler@gmail.com

Working Draft, C++ extensions for Ranges

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

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 *iterable* and *range* 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*) as arguments to the higher-order 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 (24.11).

3 This paper is based on the design presented in "Ranges for the Standard Library, Revision 1" (1.2). Refer to that document for a discussion of motivations and design trade-offs.

4 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 langauge (e.g., the numeric algorithms), nor does it add range support to all the places that could benefit from it (e.g., the containers).

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

6 The International Standard, ISO/IEC 14882, provides important context and specification for this paper. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

1.2 Normative references [intro.refs]

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

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

(1.2) — JTC1/SC22/WG21 N4377, *Technical Specification - 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 N4377 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 Acknowledgments [intro.ack]

1 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.

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*]

19 Concepts library [concepts.lib]

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.

2 The following subclauses describe core language concepts, foundational concepts, function concepts, iterator concepts, and rearrangement concepts as summarized in Table 1.

Table 1 — Fundamental concepts library summary

	Subclause	Header(s)
19.2	Core language concepts	`<concepts>`
19.3	Foundational concepts	
19.4	Function concepts	

3 In this Clause, `CamelCase` identifiers ending with “`Type`” denote template aliases.

4 A *regular function* is a function that returns equal output when passed equal input. A regular function that returns a value may copy or move the returned object, or may return a reference. Regular functions are allowed to have side effects and preconditions that, when violated, make the function non-regular. [*Note:* A function that returns `void` is necessarily a regular function. — *end note*]

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 = see below;
```

1 The `Same` concept is used when requiring that two types are the same.

2 `Same<T, U>` is an alias for `true` when `T` and `U` denote the same time after the elimination of aliases; otherwise, it is an alias for `false`. `Same<T, U>` shall have the same value as `is_same<T, U>::value`.

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

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

```
template <class T, class U>
concept bool Derived = is_base_of<U, T>::value;
```

1 The `Derived` concept is used when requiring that a type `T` is derived from `U`, or the same type as `U`.

19.2.4 Concept Convertible [concepts.lib.corelang.convertible]

```
template <class T, class U>
concept bool Convertible = is_convertible<T, U>::value;
```

1 The `Convertible` concept concept expresses the requirement that a type `T` can be implicitly converted to type `U`.

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

1 Two types `T` and `U` are *unambiguously convertible* if there exists a unique, explicit conversion from `T` to `U`, or from `U` to `T`, but not both. [*Note:* This is trivially true when `T` and `U` are the same type since no conversion would be required. — *end note*]

2 If `T` and `U` can both be unambiguously converted to a third type, `C`, `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. — *end note*] This notion is encapsulated by the `CommonType` alias and the `Common` concept.

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

template <class T, class U>
concept bool Common =
  requires (T t, U u) {
    typename CommonType<T, U>;
    {CommonType<T, U>{forward<T>(t)}};
    {CommonType<T, U>{forward<U>(u)}};
  };
```

3 Two types `T` and `U` model the `Common` concept when the type alias `CommonType<T, U>` is well-formed, and when objects of types `T` and `U` can be explicitly converted to the common type.

4 Let `C` be `CommonType<T, U>`. Let `t1` and `t2` be objects of type `T`. `t1` equals `t2` if and only if `C{t1}` equals `C{t2}`. Let `u1` and `u2` be object of type `U`. `u1` equals `u2` if and only if `C{u1}` equals `C{u2}`.

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

19.2.6 Concept Boolean [concepts.lib.corelang.boolean]

```
template <class B>
concept bool Boolean =
  requires(B b1, B b2) {
    { bool(b1) };
    { !b1 } -> Boolean;
    requires Same<decltype(b1 && b2), bool>;
    requires Same<decltype(b1 || b2), bool>;
  };
```

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

2 Given values `b1` and `b2` of type `B`, and a value `t1` of type `bool`, then type `B` models `Boolean` if and only if

(2.1) — `bool(b1) == t1` for every value `b1` implicitly converted to `t1`.

(2.2) — `bool(b1) == !bool(!b1)` for every value `b1`.

(2.3) — `(b1 && b2) == (bool(b1) && bool(b2))`.

(2.4) — `(b1 || b2) == (bool(b1) || bool(b2))`.

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

```
template <class T>
concept bool Integral = is_integral<T>::value;
```

1 The `Integral` concept is used when requiring that a type `T` is an integral type (3.9.1).

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

```
template <class T>
concept bool SignedIntegral =
  Integral<T> && is_signed<T>::value;
```

1 The `SignedIntegral` concept is used when requiring that a type `T` is an integral type that is signed.

2 [*Note:* `SignedIntegral<T>` may be true even for types that are not signed integral types (3.9.1). — *end note*]

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

```
template <class T>
concept bool UnsignedIntegral =
  Integral<T> && !SignedIntegral<T>;
```

1 The `UnsignedIntegral` concept is used when requiring that a type `T` is an integral type that is unsigned.

2 [*Note:* `UnsignedIntegral<T>` may be true even for types that are not unsigned integral types (3.9.1). — *end note*]

19.2.10 Concept DefaultConstructible [concepts.lib.corelang.defaultconstructible]

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

```
template <class T>
concept bool DefaultConstructible =
  is_default_constructible<T>::value;
```

1 A type `T` models `DefaultConstructible` if and only if

(1.1) — The definition `T t;` default-initializes (8.5) object `t`.

(1.2) — The definition `T u{};` value-initializes (8.5) object `u`.

(1.3) — The expressions `T()` and `T{}` value-initialize (8.5) temporary objects of type `T`.

[Editor's note: REVIEW: Stephan T. Lavavej wonders if we need to say anything special about aggregates.]

19.2.11 Concept MoveConstructible [concepts.lib.corelang.moveconstructible]

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

```
template <class T>
concept bool MoveConstructible =
  is_move_constructible<T>::value;
```

1 Let `rv` be an rvalue of type `T`. Then a type `T` models `MoveConstructible` 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)` is equal to the value of `rv` before the construction.

2 `rv`'s 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*]

19.2.12 Concept CopyConstructible [concepts.lib.corelang.copyconstructible]

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

```
template <class T>
concept bool CopyConstructible =
  MoveConstructible<T> && is_copy_constructible<T>::value;
```

1 Let `v` be an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`. Then a type `T` models `CopyConstructible` if and only if

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

(1.2) — `T(v)` is equal to `v` and `v` is unchanged.

19.2.13 Concept Constructible [concepts.lib.corelang.constructible]

```
template <class T, class ...Args>
concept bool Constructible =
  is_constructible<T, Args...>::value;
```

1 The `Constructible` concept is used when requiring that a type `T` is constructible with arguments of types `Args...`.

19.2.14 Concept Destructible [concepts.lib.corelang.destructible]

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

```
template <class T>
concept bool Destructible =
  is_destructible<T>::value;
```

1 A type `T` models `Destructible` if and only if

(1.1) — After the expression `u.~T()`, all resources owned by `u` are reclaimed and no exception is propagated.

19.2.15 Concept MoveAssignable [concepts.lib.corelang.moveassignable]

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

```
template <class T>
concept bool MoveAssignable =
  is_move_assignable<T>::value;
```

1 Let `rv` be an rvalue of type `T`. Then a type `T` models `MoveAssignable` if and only if

(1.1) — The expression `t = rv` is an lvalue of type (non-const) `T`, the expression returns a reference to `t`, and after the assignment `t` is equal to the value of `rv` before the assignment.

2 `rv`'s 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*]

19.2.16 Concept CopyAssignable [concepts.lib.corelang.copyassignable]

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

```
template <class T>
concept bool CopyAssignable =
  MoveAssignable<T> && is_copy_assignable<T>::value;
```

1 Let `v` be an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`. Then a type `T` models `CopyAssignable` if and only if

(1.1) — The expression `t = v` is an lvalue of type (non-`const`) `T`, the expression returns a reference to `t`, `v` is unchanged, and after the assignment `t` is equal to `v`.

19.2.17 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, class U = T>
concept bool Swappable =
  requires(T t, U u) {
    swap(forward<T>(t), forward<U>(u));
    swap(forward<U>(u), forward<T>(t));
  };
```

[Editor's note: The above requires that the `swap` function templates defined in `<utility>` are constrained to only accept movable types.] [Editor's note: The following is copied almost verbatim from [swappable.requirements]]

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 types `T` and `U` model `Swappable`. Types `T` and `U` model `Swappable` if and only if:

(2.1) — the requires clause above is evaluated in the context described below, and

(2.2) — these expressions have the following effects:

(2.2.1) — the object referred to by `t` has the value originally held by `u` and

(2.2.2) — the object referred to by `u` has the value originally held by `t`.

3 The context in which the requires clause is evaluated shall ensure that a binary non-member function named "`swap`" is selected via overload resolution (13.3) on a candidate set that includes:

(3.1) — the two `swap` function templates defined in `<utility>` (20.2) and

(3.2) — the lookup set produced by argument-dependent lookup (3.4.2).

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

[*Note:* It is unspecified whether a library component that has a swappable requirement includes the header `<utility>` to ensure an appropriate evaluation context. — *end note*]

4 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::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::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::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 Foundational concepts [concepts.lib.foundational]

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

1 This section describes the foundational concepts that describe the basis of the value-oriented programming style on which the library is based. The purpose of these concepts is to establish a foundation for equational reasoning in programs.

19.3.2 Concept EqualityComparable [concepts.lib.foundational.equalitycomparable]

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

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

1 Let `a`, `b`, and `c` be well-formed objects of type `T`. Then type `T` models `EqualityComparable` if and only if

(1.1) — `(a == a) != false`.

(1.2) — `(a == b) != false` if and only if `a` is equal to `b`.

(1.3) — `(a == b) != false` if and only if `(b == a) != false`.

(1.4) — `(a != b) != false` if and only if `(a == b) == false`.

(1.5) — `(a == b) != false && (b == c) != false` if and only if `(a == c) != false`.

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

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

3 Let `a` be an object of type `T`, `b` be an object of type `U`, and `C` be `CommonType<T, U>`. Then types `T` and `U` model `EqualityComparable` if and only if

(3.1) — `(a == b) != false` if and only if `(C{a} == C{b}) != false`.

(3.2) — `(a != b) != false` if and only if `(C{a} != C{b}) != false`.

(3.3) — `(b == a) != false` if and only if `(C{b} == C{a}) != false`.

(3.4) — `(b != a) != false` if and only if `(C{b} != C{a}) != false`.

19.3.3 Concept Semiregular [concepts.lib.foundational.semiregular]

```
template <class T>
concept bool Semiregular =
  DefaultConstructible<T> &&
  CopyConstructible<T> &&
  Destructible<T> &&
  CopyAssignable<T> &&
  requires(T a, size_t n) {
    // Object
    requires Same<T*, decltype(&a)>;
    // Destruction
    { a.~T() } noexcept;
    // Allocation
    requires Same<T*, decltype(new T)>;
    requires Same<T*, decltype(new T[n])>;
    // Deallocation
    { delete new T };
    { delete new T[n] };
  };
```

[Editor's note: The concepts are probably too fine-grained here. Look again at "The Palo Alto" report; in particular, at the definition of `Semiregular` in Appendix D.]

1 Let `a` be any well-formed object of type `T`. Then type `T` models `Semiregular` if and only if

(1.1) — `(&a == addressof(a)) != false`.

(1.2) — `a` is not a volatile object or const volatile object (3.9.3).

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

19.3.4 Concept Regular [concepts.lib.foundational.regular]

```
template <class T>
concept bool Regular =
  Semiregular<T>() && EqualitiyComparable<T>;
```

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

19.3.5 Concept TotallyOrdered [concepts.lib.foundational.totallyordered]

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

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

1 Let `a`, `b`, and `c` be well-formed objects of type `T`. Then type `T` models `TotallyOrdered` if and only if

(1.1) — `(a < a) == false`.

(1.2) — If `(a < b) != false`, then `(b < a) == false`.

(1.3) — If `(a < b) != false` and `(b < c) != false`, then `(a < c) != false`.

(1.4) — Exactly one of the following is true: `(a < b) != false`, or `(b < a) != false`, or `(a == b) != false`.

(1.5) — `(a > b) != false` if and only if `(b < a) != false`.

(1.6) — `(a <= b) != false` if and only if `(b < a) == false`.

(1.7) — `(a >= b) != false` if and only if `(b > a) == false`.

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

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

```
    TotallyOrdered<CommonType<T, U>> &&
    EqualityComparable<T, U> &&
    requires (T a, U b) {
      { a < b } -> Boolean;
      { a > b } -> Boolean;
      { a <= b } -> Boolean;
      { a >= b } -> Boolean;
      { b < a } -> Boolean;
      { b > a } -> Boolean;
      { b <= a } -> Boolean;
      { b >= a } -> Boolean;
    };
}
```

3 Let `a` be an object of type T, `b` be an object of type `U`, and `C` be `CommonType<T, U>`. Then types `T` and `U` model `EqualityComparable` if and only if

(3.1) — `(a < b) != false` if and only if `(C{a} < C{b}) != false`.

(3.2) — `(a > b) != false` if and only if `(C{a} > C{b}) != false`.

(3.3) — `(a <= b) != false` if and only if `(C{a} <= C{b}) != false`.

(3.4) — `(a >= b) != false` if and only if `(C{a} >= C{b}) != false`.

(3.5) — `(b < a) != false` if and only if `(C{b} < C{a}) != false`.

(3.6) — `(b > a) != false` if and only if `(C{b} > C{a}) != false`.

(3.7) — `(b <= a) != false` if and only if `(C{b} <= C{a}) != false`.

(3.8) — `(b >= a) != false` if and only if `(C{b} >= C{a}) != false`.

19.4 Function concepts [concepts.lib.functions]

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

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

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

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

template <class F, class...Args>
concept bool Function =
  Destructible<T> &&
  CopyConstructible<T> &&
  requires (F f, Args...args) {
    typename ResultType<F, Args...>;
    requires Same<F*, decltype(&f)>;
    { f.~F() } noexcept;
    requires Same<F*, decltype(new F)>;
    { delete new F };
    { f(forward<Args>(args)...) };
    requires Same<ResultType<F, Args...>,
                  decltype(f(forward<Args>(args)...))>;
  };
```

1 Let `f` be any object of type `F`. Then types `F` and `Args...` model `Function` if and only if

(1.1) — `(&f == addressof(f)) != false`.

2 The function call expression need not be equality-preserving; that is, the expression `f(args...)` is allowed to return non-equal objects when called with equal arguments, and still model `Function`.

3 [*Note:* Since models of `Function` are allowed to be non-equality-preserving, a function that generates random numbers may model `Function`. — *end note*]

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

```
template <class F, class...Args>
concept bool RegularFunction =
  Function<F, Args...>;
```

1 Models of `RegularFunction` shall be equality-preserving; that is, when passed equal arguments, they shall return equal objects.

2 [*Note:* A random number generator is not a model of `RegularFunction`. — *end note*]

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

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

```
template <class F, class...Args>
concept bool Predicate =
  RegularFunction<F, Args...> &&
  Convertible<ResultType<F, Args...>, bool>;
```

19.4.5 Concept Relation [concepts.lib.functions.relation]

```
template <class F, class T>
concept bool Relation() {
  return Predicate<F, T, T>;
}

template <class R, class T, class U>
concept bool Relation() {
  return Relation<R, T>() &&
    Relation<R, U>() &&
    Common<T, U> &&
    Relation<R, CommonType<T, U>>() &&
    requires (R r, T a, U b) {
      { r(a, b) } -> Boolean;
      { r(b, a) } -> Boolean;
    };
}
```

1 Let `r` be any well-forned object of type `R`, `a` be any well-formed object of type `T`, `b` be any well-formed object of type `U`, and `C` be `CommonType<T, U>`. Then types `R`, `T`, and `U` model `Relation` if and only if

(1.1) — `r(a, b) != false` if and only if `r(C{a}, C{b}) != false`.

(1.2) — `r(b, a) != false` if and only if `r(C{b}, C{a}) != false`.

20 General utilities library [utilities]

20.9 Function objects [function.objects]

[Editor's note: To `<functional>` header synopsis, add `identity` function object.]

2 **Header `<functional>` synopsis**

```
struct identity;
```

[Editor's note: Add constraints to the comparison function objects.]

20.9.6 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 TotallyOrdered<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 TotallyOrdered<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 TotallyOrdered<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 TotallyOrdered<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~~bool 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~~bool 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 TotallyOrdered<T, U>()
  constexpr ~~auto~~bool 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 TotallyOrdered<T, U>()
  constexpr ~~auto~~bool 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 TotallyOrdered<T, U>()
  constexpr autobool 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 TotallyOrdered<T, U>()
  constexpr autobool 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;
};
```

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>.]

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 (24.10.2.4).

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

Table 2 — Iterators library summary

	Subclause	Header(s)
24.2	Requirements	
24.7	Iterator primitives	`<iterator>`
24.8	Predefined iterators	
24.9	Stream iterators	
24.10	Iterables	

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 (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 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~~seven categories of iterators, according to the operations defined on them: *weak input iterators,* *input iterators*, *weak output iterators,* *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 3.

Table 3 — Relations among iterator categories

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

3 The seven categories of iterators correspond to the iterator concepts `WeakInputIterator`, `InputIterator`, `WeakOutputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator`, and `RandomAccess-Iterator`, respectively. The generic term *iterator* refers to any type that models `WeakIterator`.

4 ~~Forward~~Input iterators satisfy all the requirements of weak input iterators and can be used whenever ~~an~~a weak input iterator is specified; Forward iterators also satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements

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

5 Iterators that further satisfy the requirements of weak output iterators are called *mutable iterator*s. Non-mutable 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 An iterator or sentinel `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == j`. If `j` is reachable from `i`, they refer to elements of the same sequence.

8 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators or an iterator and a sentinel that designate the beginning and end of the computation. A range `[i,i)` is an empty range; in general, a range `[i,j)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element ~~pointed to~~denoted by `j`. Range `[i,j)` is valid if and only if `j` is reachable from `i`. The result of the application of functions in the library to invalid ranges is undefined.

9 A *sentinel* is an abstraction of a past-the-end iterator. Sentinels are `Regular` types that can be used to denote the end of a range. A sentinel and an iterator denoting a range shall be `EqualityComparable`. A sentinel denotes an element when an iterator `i` compares equal to the sentinel, and `i` points to that element.

10 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.~~

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

12 An *invalid* iterator is an iterator that may be singular.[1]

13 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`~~`DifferenceType<X>` and ~~`iterator_traits<X>::reference`~~ `ReferenceType<X>`, 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 weak output iterator. [*Note:* For an iterator type `X` ~~there must be an instantiation of `iterator_traits<X>` (24.4.1)~~the type aliases `DifferenceType<X>` and `ReferenceType<X>` must be well-formed. — *end note*]

24.2.2 Readable types [readable.iterators]

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

1) 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 =
  Semiregular<I> &&
  requires (I i) {
    typename ValueType<I>;
    { *i } -> const ValueType<I>&; // pre: i is dereferencable
  };
```

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

```
template <class, class = void> struct value_type { };
template <class T>
struct value_type<T*>
  : enable_if<!is_void<T>::value, remove_cv_t<T>> { };
template <class T>
struct value_type<T[]> : remove_cv<T> { };
template <class T, size_t N>
struct value_type<T[N]> : remove_cv<T> { };
template <class T>
struct value_type<T, void_t<typename T::value_type>>
  : enable_if<!is_void<typename T::value_type>::value, typename T::value_type> { };
template <class T>
struct value_type<T, void_t<typename T::element_type>>
  : enable_if<!is_void<typename T::element_type>::value, typename T::element_type> { };

template <class T>
using ValueType = 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` that has a nested type `value_type`, `value_type<I>::type` names that type, unless it is `void` 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` that has a nested type `element_type`, `value_type<I>::type` names that type, unless it is `void` 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 MoveWritable types [movewritable.iterators]

1 The `MoveWritable` concept describes the requirements for moving a value into an iterator's referenced object.

```
template <class Out, class T>
concept bool MoveWritable =
  Semiregular<Out> &&
  requires (Out o, T v) {
    { *o = move(v) };
  };
```

2 Let `v` be an rvalue or a lvalue of type (possibly `const`) `T`, and let `o` be an object of type `Out`. Then types `T` and `Out` model `MoveWritable` if and only if

(2.1) — If `o` is dereferencable, then after the assignment `*o = move(v)`, the value referred to by `*o` is equal to the value of `v` before the assignment.

3 After the expression `*o = move(v)`, object `o` is not required to be dereferencable.

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

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 Writable types [writable.iterators]

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

```
template <class Out, class T>
concept bool Writable =
  Semiregular<Out> &&
  MoveWritable<Out, T> &&
  requires (Out o, T v) {
    { *o = v };
  };
```

2 Let `v` be an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`, and let `o` be an object of type `Out`. Then types `T` and `Out` model `Writable` if and only if

(2.1) — After the assignment `*o = v`, the value referred to by `*o` is equal to the value of `v` and `v` is unchanged.

24.2.5 IndirectlyMovable types [indirectlymovable.iterators]

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

```
template <class I, class Out>
concept bool IndirectlyMovable =
  Readable<I> &&
  Semiregular<Out> &&
  MoveWritable<Out, ValueType<I>>;
```

2 Let `i` be an object of type `I`, and let `o` be an object of type `Out`. Then types `I` and `Out` model `IndirectlyMovable` if and only if

(2.1) — After the assignment `*o = move(*i)`, the value referred to by `*o` is equal to the value of `*i` before the assignment.

24.2.6 IndirectlyCopyable types [indirectlycopyable.iterators]

1 The `IndirectlyCopyable` concept describes the copy relationship between a `Readable` type and a `Writable` type.

```
template <class I, class Out>
concept bool IndirectlyCopyable =
  Readable<I> &&
  Semiregular<I> &&
  IndirectlyMovable<I, Out> &&
  Writable<Out, ValueType<I>>;
```

2 Let `i` be an object of type `I`, and let `o` be an object of type `Out`. Then types `I` and `Out` model `IndirectlyCopyable` if and only if

(2.1) — After the assignment `*o = *i`, the value referred to by `*o` is equal to the value of `*i` and the value of `*i` is unchanged.

24.2.7 IndirectlySwappable types [indirectlyswappable.iterators]

1 The `IndirectlySwappable` concept describes a swappable relationship between the value types of two `Readable` types.

```
template <class I1, class I2 = I1>
concept bool IndirectlySwappable =
  Readable<I1> &&
  Readable<I2> &&
  Swappable<ReferenceType<I1>, ReferenceType<I2>> &&
  Swappable<ReferenceType<I1>> &&
  Swappable<ReferenceType<I2>>;
```

24.2.8 WeaklyIncrementable types [weaklyincrementable.iterators]

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

```
template <class I>
concept bool WeaklyIncrementable =
  Semiregular<I> &&
  requires (I i) {
    typename DifferenceType<I>;
    requires SignedIntegral<DifferenceType<I>>;
    { ++i };
    requires Same<I&, decltype(++i)>;
    { i++ };
  };
```

1 Not all arguments will be incrementable for a given type. For example NaN is not a well-formed floating point value and hence is not incrementable. Likewise, past-the-end iterators are also not incrementable. This does not mean that the type does not model `WeaklyIncrementable`.

2 Let `i` be an object of type `I`. Then the type `I` models `WeaklyIncrementable` if and only if

(2.1) — If `i` is incrementable, then `++i` moves `i` to the next element.

(2.2) — If `i` is incrementable, then `(&++i == &i) != false`.

(2.3) — If `i` is incrementable, then `i++` moves `i` to the next element.

(2.4) — `++i` is valid if and only if `i++` is valid.

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

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

24.2.9 Incrementable types [incrementable.iterators]

The `Incrementable` concept describes types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be `EqualityComparable`.

```
template <class I>
concept bool Incrementable =
  Regular<I> &&
  WeaklyIncrementable<I> &&
  Same<I, decltype(i++)>;
```

1 Let `a` and `b` be incrementable objects of type `I`. Then the type `I` models `Incrementable` if and only if

(1.1) — If `(a == b) != false` then `(++a == ++b) != false`.

(1.2) — If `(a == b) != false` then `(a++, a) == (b++, b) != false`.

(1.3) — If `(a == b) != false` then `(a++ == b) != false`.

(1.4) — If `(a == b) != false` then `((a++, a) == ++b) != false`.

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

2 [*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 model `Incrementable`. — *end note*]

24.2.10 Weak iterators [weakiterator.iterators]

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

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

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

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

24.2.11 Iterators [iterator.iterators]

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

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

```
template <class I>
concept bool Iterator =
  WeakIterator<I> &&
  EqualityComparable<I>();
```

24.2.12 Sentinels [sentinel.iterators]

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

```
template <class T, class I>
concept bool Sentinel =
  Regular<T> &&
  Iterator<I> &&
  EqualityComparable<T, I>();
```

24.2.13 Weak input iterators [weakinput.iterators]

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

```
template <class I>
concept bool WeakInputIterator =
  WeakIterator<I> &&
  Readable<I> &&
  requires(I i) {
    typename IteratorCategory<I>;
    { i++ } -> Readable;
    requires Derived<IteratorCategory<I>, weak_input_iterator_tag>;
  };
```

24.2.14 Input iterators [input.iterators]

[Editor's note: Remove para 1, 2 and Table 107]

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

```
template <class I>
concept bool InputIterator =
  WeakInputIterator<I> &&
  Iterator<I> &&
  Derived<IteratorCategory<I>, input_iterator_tag>;
```

2 [*Note:* For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Value type T is not required to be a `CopyAssignable` type (~~Table~~ 19.2.16). These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class template. — *end note*]

[Editor's note: Section Output iterators renamed to Weak output iterators below:]

24.2.15 Weak output iterators [weakoutput.iterators]

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

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

```
template <class I, class T>
concept bool WeakOutputIterator =
  WeakIterator<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 weak 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*]

24.2.16 Output iterators [output.iterators]

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

```
template <class I, class T>
concept bool OutputIterator =
  WeakOutputIterator<I, T> && Iterator<I>;
```

2 [*Note:* Output iterators are used by single-pass algoritms that write into a bounded range, like `generate`. — *end note*]

24.2.17 Forward iterators [forward.iterators]

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 (24.2.14),

(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` (24.2.14) and adds the multi-pass guarantee, described below.

```
template <class I>
concept bool ForwardIterator =
  InputIterator<I> &&
  Incrementable<I> &&
  Derived<IteratorCategory<I>, forward_iterator_tag>;
```

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

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 the expression `(void)++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 iterators) and the removal of the restrictions on the number of the assignments through a mutable iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

[Editor's note: Remove Table 109]

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.

24.2.18 Bidirectional iterators [bidirectional.iterators]

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` (24.2.17), and adds the ability to move an iterator backward as well as forward.

```
template <class I>
concept bool BidirectionalIterator =
  ForwardIterator<I> &&
  Derived<IteratorCategory<I>, bidirectional_iterator_tag> &&
  requires (I i) {
    { --i };
    requires Same<I&, decltype(--i)>;
    { i-- };
    requires Same<I, decltype(i--)>;
  };
```

[Editor's note: Remove table 110]

3 A bidirectional iterator `r` is decrementable 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 `I` models `BidirectionalIterator` if and only if:

(4.1) — `&--a == &a`.

(4.2) — If `(a == b) != false`, then `(a-- == j) != false`.

(4.3) — If `(a == b) != false`, then `((a--, a) == --j) != false`.

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

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

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

24.2.19 Random access iterators [random.access.iterators]

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

```
template <class I>
concept bool RandomAccessIterator =
  BidirectionalIterator<I> &&
  TotallyOrdered<I>() &&
  Derived<IteratorCategory<I>, random_access_iterator_tag> &&
  SizedIteratorRange<I, I> && // see below
  requires (I i, I j, DifferenceType<I> n) {
    { i += n };
    { i + n };
    { n + i };
    { i -= n };
    { i - n };
```

```
    requires Same<decltype(i += n), I&>;
    requires Same<decltype(i + n), I>;
    requires Same<decltype(n + i), I>;
    requires Same<decltype(i -= n), I&>;
    requires Same<decltype(i - n), I>;
    { i[n] } -> const ValueType<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 an object of type `DifferenceType<I>` such that after `n` applications of `++a`, `(a == b) != false`. Then `I` models `RandomAccessIterator` if and only if:

(2.1) — `(a += n) == b`.

(2.2) — `&(a += n) == &a`.

(2.3) — `(a + n) == (a += n)`.

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

(2.5) — `a + 0 == a`.

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

(2.7) — `(b += -n) == a`.

(2.8) — `(b -= n) == a`.

(2.9) — `&(b -= n) == &b`.

(2.10) — `(b - n) == (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.3).]

24.3.2 Function type [functiontype.indirectcallables]

1 The `FunctionType` is an alias used to turn a callable type (20.9.1) into a function object type (20.9).

```
// Exposition only
template <class T>
auto __as_fun_obj(T&& t, std::true_type) {
  return mem_fn(t);
}
template <class T>
T __as_fun_obj(T&& t, std::false_type) {
  return forward<T>(t);
}
```

```
template <class T>
using FunctionType =
  decltype(__as_fun_obj(declval<T>(), is_member_pointer<decay_t<T>>{}));
```

24.3.3 Projected iterator [projected.indirectcallables]

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 projection function `Proj` into a new `Readable` type whose `reference` type is the result of applying `Proj` to the `ReferenceType` of `I`.

```
template <Readable I, class Proj>
  requires RegularFunction<FunctionType<Proj>, ValueType<I>>
struct Projected {
  using difference_type = DifferenceType<I>;
  using value_type = decay_t<ResultType<FunctionType<Proj>, ValueType<I>>>;
  ResultType<FunctionType<Proj>, ReferenceType<I>> operator*() const;
};
```

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

24.3.4 Indirect callables [indirectfunc.indirectcallables]

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

template <class F, class...Is>
concept bool IndirectCallable =
  (Readable<Is> && ...) &&
  Function<FunctionType<F>, ValueType<Is>...>;

template <class F, class...Is>
concept bool IndirectRegularCallable =
  (Readable<Is> && ...) &&
  RegularFunction<FunctionType<F>, ValueType<Is>...>;

template <class F, class...Is>
concept bool IndirectCallablePredicate =
  (Readable<Is> && ...) &&
  Predicate<FunctionType<F>, ValueType<Is>...>;

template <class F, class I1, class I2 = I1>
concept bool IndirectCallableRelation =
  Readable<I1> &&
  Readable<I2> &&
  Relation<FunctionType<F>, ValueType<I1>, ValueType<I2>>();
```

24.4 Common algorithm requirements [commmonalgoreq]

24.4.1 In general [commmonalgoreq.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 four relational concepts for rear-

rangements: `Permutable`, `Mergeable`, `MergeMovable`, and `Sortable`. There is one relational concept for comparing values from different sequences: `IndirectlyComparable`.

24.4.2 Indirectly comparable iterators [indirectlycomparable.commmonalgoreq]

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 =
  IndirectCallableRelation<R, Projected<I1, P1>, Projected<I2, P2>>;
```

24.4.3 Permutable iterators [permutable.commmonalgoreq]

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

```
template <class I>
concept bool Permutable =
  ForwardIterator<I> &&
  Semiregular<ValueType<I>> &&
  IndirectlyMovable<I>;
```

[Editor's note: REVIEW: `Semiregular` here overconstrains by adding a default-constructibility requirement. See Appendix D of the "The Palo Alto" report for an alternate design.]

24.4.4 Mergeable iterators [mergeable.commmonalgoreq]

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

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

2 The relation `R` must represent a strict weak ordering.

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

24.4.5 MergeMovable iterators [mergemovable.commmonalgoreq]

1 The `MergeMovable` concept describes the requirements of algorithms that merge sorted sequences into an output sequence by moving elements.

```
template <class I1, class I2, class Out,
    class R = less<>, class P1 = identity, class P2 = identity>
concept bool MergeMovable =
  InputIterator<I1> &&
  InputIterator<I2> &&
  WeaklyIncrementable<Out> &&
  IndirectlyMovable<I1, Out> &&
  IndirectlyMovable<I2, Out> &&
  IndirectlyComparable<I1, I2, R, P1, P2>;
```

2 The relation `R` must represent a strict weak ordering.

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

24.4.6 Sortable iterators [sortable.commmonalgoreq]

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

```
template <class I, class R = less<>, class P = identity>
concept bool Sortable =
  ForwardIterator<I> &&
  Permutable<I> &&
  IndirectCallableRelation<R, Projected<I, P>>;
```

2 The relation `R` must represent a strict weak ordering.

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

24.5 Iterator range requirements [iteratorranges]

24.5.1 Sized Iterator range [sizediteratorrange.iteratorranges]

The `SizedIteratorRange` concept describes the requirements on an `Iterator` (24.2.11) and a `Sentinel` that allows the use of the `-` operator to compute the distance between them in constant time.

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

1 Let `a` be a valid iterator of type `I` and `b` be a valid sentinel of type `S`. Let `n` be an object of type `DifferenceType<I>` such that after `n` applications of `++a`, `(a == b) != false`. Then types `I` and `S` model `SizedIteratorRange` if and only if:

(1.1) — `(b - a) == n`.

(1.2) — `(a - b) == -n`.

(1.3) — `(a - a) == 0`.

(1.4) — `(b - b) == 0`.

[*Note:* The `SizedIteratorRange` concept is modeled by pairs of `RandomAccessIterators` (24.2.19) and by counted iterators and their sentinels (24.8.5.1). — *end note*]

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

24.6 Header <iterator> synopsis [iterator.synopsis]

```
namespace std {
  // 24.7, primitives:
  template<class Iterator> ~~struct~~using iterator_traits = see below;
  ~~template<class T> struct iterator_traits<T*>;~~
```

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

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

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

// 24.7.4, iterator operations:

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

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

// 24.8, predefined iterators and sentinels:
```

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

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

template <~~class Iterator1~~BidirectionalIterator I1, ~~class Iterator2~~BidirectionalIterator I2>
    requires SizedIteratorRange<I2, I1>
  ~~auto~~DifferenceType<I2> 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~~DifferenceType<I> n,
  const reverse_iterator<~~Iterator~~I>& x);

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

// 24.8.2 Insert iterators
template <class Container> class back_insert_iterator;
template <class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;
template <class Container>
```

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

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

// 24.8.3 Move iterators
template <~~class Iterator~~WeakInputIterator I> class move_iterator;
template <~~class Iterator1~~InputIterator I1, ~~class Iterator2~~InputIterator I2>
    requires EqualityComparable<I1, I2>()
  bool operator==(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <~~class Iterator1~~InputIterator I1, ~~class Iterator2~~InputIterator I2>
    requires EqualityComparable<I1, I2>()
  bool operator!=(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator<(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator<=(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator>(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);
template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator>=(
    const move_iterator<~~Iterator1~~I1>& x, const move_iterator<~~Iterator2~~I2>& y);

template <~~class Iterator1~~WeakInputIterator I1, ~~class Iterator2~~WeakInputIterator I2>
    requires SizedIteratorRange<I2, I1>
  ~~auto~~DifferenceType<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~~DifferenceType<I> n,
  const move_iterator<~~Iterator~~I>& x);
template <~~class Iterator~~WeakInputIterator I>
  move_iterator<~~Iterator~~I> make_move_iterator(~~Iterator~~I i);

// 24.8.4 Common iterators
template<class A, classB>
concept bool __WeaklyEqualityComparable;   // exposition only
template<class I, class S>
concept bool __WeakSentinel;               // exposition only

template <InputIterator I, __WeakSentinel<I> S> class common_iterator;
template <InputIterator I1, __WeakSentinel<I1> S1,
```

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

template <InputIterator I1, __WeakSentinel<I1> S1,
          InputIterator I2, __WeakSentinel<I2> S2>
  requires SizedIteratorRange<I1, I1> && SizedIteratorRange<I2, I2> &&
    requires (I1 a, I2 b) { {a-b}->DifferenceType<I2>; {b-a}->DifferenceType<I2>; }
    requires (I1 i, S2 s) { {i-s}->DifferenceType<I2>; {s-i}->DifferenceType<I2>; }
    requires (I2 i, S1 s) { {i-s}->DifferenceType<I2>; {s-i}->DifferenceType<I2>; }
DifferenceType<I2> operator-(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);

// 24.8.5 Counted iterators and sentinels
[Editor's note:  TODO: Why not WeakIterator?]
template <WeakInputIterator I> class counted_iterator;
class counted_sentinel;

template <WeakInputIterator I1, WeakInputIterator I2>
  bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <WeakInputIterator I>
  bool operator==(
    const counted_iterator<I>& x, counted_sentinel y);
template <WeakInputIterator I>
  bool operator==(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator==(counted_sentinel x, counted_sentinel y);
template <WeakInputIterator I1, WeakInputIterator I2>
  bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <WeakInputIterator I>
  bool operator!=(
    const counted_iterator<I>& x, counted_sentinel y);
template <WeakInputIterator I>
  bool operator!=(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator!=(counted_sentinel x, counted_sentinel y);

template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator<(
    const counted_iterator<I>& x, counted_sentinel y);
template <RandomAccessIterator I>
```

```
  bool operator<(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator<(counted_sentinel x, counted_sentinel y);
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator<=(
    const counted_iterator<I>& x, counted_sentinel y);
template <RandomAccessIterator I>
  bool operator<=(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator<=(counted_sentinel x, counted_sentinel y);
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator>(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator>(
    const counted_iterator<I>& x, counted_sentinel y);
template <RandomAccessIterator I>
  bool operator>(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator>(counted_sentinel x, counted_sentinel y);
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator>=(
    const counted_iterator<I>& x, counted_sentinel y);
template <RandomAccessIterator I>
  bool operator>=(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator>=(counted_sentinel x, counted_sentinel y);

template <WeakInputIterator I1, WeakInputIterator I2>
  DifferenceType<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <WeakInputIterator I>
  DifferenceType<I> operator-(
    const counted_iterator<I>& x, counted_sentinel y);
template <WeakInputIterator I>
  DifferenceType<I> operator-(
    counted_sentinel x, const counted_iterator<I>& y);
ptrdiff_t operator-(counted_sentinel x, counted_sentinel y);
template <RandomAccessIterator I>
  counted_iterator<I>
    operator+(DifferenceType<I> n, const counted_iterator<I>& x);
template <WeakInputIterator I>
  counted_iterator<I> make_counted_iterator(I i, DifferenceType<I> n);

template <WeakInputIterator I>
  void advance(counted_iterator<I>& i, DifferenceType<I> n);
```

```
// 24.8.5.4 common_type specializations
template<WeakInputIterator I>
  struct common_type<counted_iterator<I>, counted_sentinel>;
template<WeakInputIterator I>
  struct common_type<counted_sentinel, counted_iterator<I>>;

// 24.8.6 Unreachable sentinels
struct unreachable { };
template <Iterator I>
  constexpr bool operator==(I const &, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator==(unreachable, I const &) noexcept;
constexpr bool operator==(unreachable, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator!=(I const &, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator!=(unreachable, I const &) noexcept;
constexpr bool operator!=(unreachable, unreachable) noexcept;

// 24.8.6.3 common_type specializations
template<Iterator I>
  struct common_type<I, unreachable>;
template<Iterator I>
  struct common_type<unreachable, I>;


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

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

// 24.11, range access:
template <class C> auto begin(C& c) -> decltype(c.begin());
```

```
template <class C> auto begin(const C& c) -> decltype(c.begin());
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
template <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));

template <class C> auto size(const C& c) -> decltype(c.size());
template <class T, size_t N> constexpr size_t begin(T (&array)[N]) noexcept;
template <class E> size_t size(initializer_list<E> il) noexcept;

}
```

24.7 Iterator primitives [iterator.primitives]

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

24.7.1 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~~`WeaklyIncrementable` is the name of a type that models the WeaklyIncrementable concept(24.2.8), `Readable` is the name of a type that models the Readable concept(24.2.2), and `WeakInputIterator` is the name of a type that models the WeakInputIterator(24.2.13) concept, the types

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

```
DifferenceType<WeaklyIncrementable>
ValueType<Readable>
IteratorCategory<WeakInputIterator>
```

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

```
ReferenceType<Readable>
```

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

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

```
  template <class, class = void> struct difference_type { };
  template <class T> struct difference_type<T*> {
    using type = ptrdiff_t;
  };
  template <> struct difference_type<nullptr_t> {
    using type = ptrdiff_t;
  };
  template <class T> struct difference_type<T[]> {
    using type = ptrdiff_t;
  };
  template <class T, size_t N> struct difference_type<T[N]> {
    using type = ptrdiff_t;
  };
```

```
template <class T>
struct difference_type<T, void_t<typename T::difference_type>> {
  using type = typename T::difference_type;
}
template <class T>
struct difference_type<T, enable_if_t<is_integral<T>::value>>
  : make_signed< decltype(declval<T>() - declval<T>()) > {
};
template <class T>
  using DifferenceType = typename difference_type<T>::type;
```

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

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

6 `IteratorCategory<T>` is implemented as if:

```
template <class, class = void> struct iterator_category { };
template <class T> struct iterator_category<T*> {
  using type = random_access_iterator_tag;
};
template <class T>
struct iterator_category<T, void_t<typename T::iterator_category>> {
  using type = typename T::iterator_category;
};
template <class T>
  using IteratorCategory = typename iterator_category<T>::type;
```

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

8 [*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*]

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

```
template <WeakInputIterator I, class = void> struct __pointer_type {
  using type = add_pointer_t<ReferenceType<I>>;
};
template <WeakInputIterator I>
struct __pointer_type<I, void_t<decltype(declval<I>().operator->())>> {
  using type = decltype(declval<I>().operator->());
};
template <class> struct __iterator_traits { };
template <WeakIterator I> struct __iterator_traits<I> {
```

```
    using difference_type = DifferenceType<I>;
    using value_type = void;
    using reference = void;
    using pointer = void;
    using iterator_category = output_iterator_tag;
  };
  template <WeakInputIterator I> struct __iterator_traits<I> {
    using difference_type = DifferenceType<I>;
    using value_type = ValueType<I>;
    using reference = ReferenceType<I>;
    using pointer = typename __pointer_type<I>::type;
    using iterator_category = IteratorCategory<I>;
  };
  template <class I>
    using iterator_traits = __iterator_traits<I>;
```

10 [*Note:* `iterator_traits` is a template alias to intentionally break code that tries to specialize it. — *end note*]

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

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

— *end example*]

24.7.2 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;
  };
}
```

2 [*Note:* The `Pointer` and `Reference` template parameters, and the nested `pointer` and `reference` type aliases are for backward compatibility only; they are never used by any other part of this standard. — *end note*]

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

```
namespace std {
  struct weak_input_iterator_tag { };
  struct input_iterator_tag: public weak_input_iterator_tag { };
  struct output_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~~one possible implementation is as follows:

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

```
}

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`~~`difference_type`, `value_type`, or `iterator_category` templates. — *end example*]

24.7.4 Iterator operations [iterator.operations]

1 Since only ~~random access iterators~~models of `RandomAccessIterator` provide the `+` ~~and~~operator, and models of `SizedIteratorRange` provide the `-` operator~~s~~, the library provides two function templates `advance` and `distance`. These function templates use `+` and `-` for random access iterators and sized iterator ranges, respectively (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations.

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

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

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

3 *Effects:* Increments (or decrements for negative `n`) iterator reference `i` by `n`.

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

4 *Requires:* `bound` shall be reachable from `i`.

5 *Effects:* Increments iterator reference `i` until `i == bound`.

6 If `I` and `S` are the same type, this function is constant time.

7 If `I` and `S` model the concept `SizedIteratorRange`, this function shall dispatch to `advance(i, bound - i)`.

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

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

9 *Effects:* Increments (or decrements for negative `n`) iterator reference `i` either `n` times or until `i == bound`, whichever comes first.

10 If `I` and `S` model `SizedIteratorRange`:

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

(10.2) — Otherwise, this function dispatches to `advance(i, n)`.

11 *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>
  DifferenceType<I> distance(I first, S last);
```

12 *Effects:* If ~~`InputIterator`~~I and S ~~meets the requirements of random access iterator~~model `SizedIteratorRange`, returns `(last - first)`; otherwise, returns the number of increments needed to get from `first` to `last`.

13 *Requires:* If ~~`InputIterator`~~I and S ~~meets the requirements of random access iterator~~model `SizedIteratorRange`, `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 <WeakIterator I>
  I next(I x, DifferenceType<I> n = 1);
```

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

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

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

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

16 *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, DifferenceType<I> n = 1);
```

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

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

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

24.8 Iterator adaptors [predef.iterators]

24.8.1 Reverse iterators [reverse.iterators]

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)`.

24.8.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  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 = DifferenceType<I>;
    using value_type = ValueType<I>;
    using iterator_category = IteratorCategory<I>;
    using reference = ReferenceType<I>;

    reverse_iterator();
    explicit reverse_iterator(~~Iterator~~I x);
    template <~~class~~BidirectionalIterator U>
      requires Convertible<U, I>
    reverse_iterator(const reverse_iterator<U>& u);
    template <~~class~~BidirectionalIterator U>
      requires Convertible<U, I>
    reverse_iterator& operator=(const reverse_iterator<U>& u);

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

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

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

```
        requires RandomAccessIterator<I>;
      reverse_iterator& operator-=(difference_type n)~~;~~
        requires RandomAccessIterator<I>;
      unspecified operator[](difference_type n) const~~;~~
        requires RandomAccessIterator<I>
    protected:
      ~~Iterator~~I current;
    };

    template <~~class Iterator1~~BidirectionalIterator I1, ~~class Iterator2~~BidirectionalIterator I2>
        requires EqualityComparable<I1, I2>()
      bool operator==(
        const reverse_iterator<~~Iterator1~~I1>& x,
        const reverse_iterator<~~Iterator2~~I2>& y);
    template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
        requires TotallyOrdered<I1, I2>()
      bool operator<(
        const reverse_iterator<~~Iterator1~~I1>& x,
        const reverse_iterator<~~Iterator2~~I2>& y);
    template <~~class Iterator1~~BidirectionalIterator I1, ~~class Iterator2~~BidirectionalIterator I2>
        requires EqualityComparable<I1, I2>()
      bool operator!=(
        const reverse_iterator<~~Iterator1~~I1>& x,
        const reverse_iterator<~~Iterator2~~I2>& y);
    template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
        requires TotallyOrdered<I1, I2>()
      bool operator>(
        const reverse_iterator<~~Iterator1~~I1>& x,
        const reverse_iterator<~~Iterator2~~I2>& y);
    template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
        requires TotallyOrdered<I1, I2>()
      bool operator>=(
        const reverse_iterator<~~Iterator1~~I1>& x,
        const reverse_iterator<~~Iterator2~~I2>& y);
    template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
        requires TotallyOrdered<I1, I2>()
      bool operator<=(
        const reverse_iterator<~~Iterator1~~I1>& x,
        const reverse_iterator<~~Iterator2~~I2>& y);
    template <~~class Iterator1~~BidirectionalIterator I1, ~~class Iterator2~~BidirectionalIterator I2>
        requires SizedIteratorRange<I2, I1>
      ~~auto~~DifferenceType<I2> 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~~DifferenceType<I> n,
      const reverse_iterator<~~Iterator~~I>& x);

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

24.8.1.2 `reverse_iterator` requirements [reverse.iter.requirements]

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

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

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

24.8.1.3.1 `reverse_iterator` constructor [reverse.iter.cons]

```
reverse_iterator();
```

1 *Effects:* Value initializes `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type ~~`Iterator`~~`I`.If `I` is a literal type, then this constructor shall be a trivial constructor.

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

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

```
template <~~class~~BidirectionalIterator U>
  requires Convertible<U, I>
reverse_iterator(const reverse_iterator<U>& u);
```

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

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

```
template <~~class~~BidirectionalIterator U>
  requires Convertible<U, I>
reverse_iterator&
  operator=(const reverse_iterator<U>& u);
```

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

2 *Returns:* `*this`.

24.8.1.3.3 Conversion [reverse.iter.conv]

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

1 *Returns:* `current`.

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

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

1 *Effects:*

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

24.8.1.3.5 `operator->` [reverse.iter.opref]

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

1 *Returns:* `std::addressof(operator*())`.

24.8.1.3.6 `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;
```

24.8.1.3.7 `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;
```

24.8.1.3.8 `operator+` [reverse.iter.op+]

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

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

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

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

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

2 *Returns:* `*this`.

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

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

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

24.8.1.3.11 operator-= [reverse.iter.op-=]

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

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

2 *Returns:* `*this`.

24.8.1.3.12 operator[] [reverse.iter.opindex]

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

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

24.8.1.3.13 operator== [reverse.iter.op==]

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

1 *Returns:* `x.current == y.current`.

24.8.1.3.14 operator< [reverse.iter.op<]

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

1 *Returns:* `x.current > y.current`.

24.8.1.3.15 operator!= [reverse.iter.op!=]

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

1 *Returns:* `x.current != y.current`.

24.8.1.3.16 operator> [reverse.iter.op>]

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

1 *Returns:* `x.current < y.current`.

24.8.1.3.17 operator>= [reverse.iter.op>=]

```
template <~~class Iterator1~~RandomAccessIterator I1, ~~class Iterator2~~RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
```

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

1 *Returns:* `x.current <= y.current`.

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

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

1 *Returns:* `x.current >= y.current`.

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

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

1 *Returns:* `y.current - x.current`.

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

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

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

24.8.1.3.21 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)`.

24.8.2 Insert iterators [insert.iterators]

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. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where

the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

24.8.2.1 Class template `back_insert_iterator` [back.insert.iterator]

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

```
namespace std {
  template <class Container>
  class back_insert_iterator :{
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;

  public:
    typedef Containerusing container_type = Container;
    using difference_type = ptrdiff_t;
    using iterator_category = output_iterator_tag;
    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);
}
```

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

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

`back_insert_iterator();`

1 *Effects:* Value-initializes `container`. This constructor shall be a trivial constructor.

`explicit back_insert_iterator(Container& x);`

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

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

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

24.8.2.2.5 `back_inserter` [back.inserter]

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

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

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

```
namespace std {
  template <class Container>
  class front_insert_iterator ~~:~~{
    ~~public iterator<output_iterator_tag,void,void,void,void> {~~
  protected:
    Container* container;

  public:
    ~~typedef Container~~using container_type = Container;
    using difference_type = ptrdiff_t;
    using iterator_category = output_iterator_tag;
    front_insert_iterator();
    explicit front_insert_iterator(Container& x);
    front_insert_iterator<Container>&
      operator=(const typename Container::value_type& value);
    front_insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

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

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

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

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

```
front_insert_iterator();
```

1 *Effects:* Value-initializes `container`. This constructor shall be a trivial constructor.

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

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

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

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

24.8.2.4.5 `front_inserter` [front.inserter]

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

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

24.8.2.5 Class template `insert_iterator` [insert.iterator]

```
namespace std {
  template <class Container>
  class insert_iterator ~~:~~{
    ~~public iterator<output_iterator_tag,void,void,void,void> }~~
  protected:
    Container* container;
    typename Container::iterator iter;

  public:
    ~~typedef Container~~using container_type = Container;
    using difference_type = ptrdiff_t;
    using iterator_category = output_iterator_tag;
    insert_iterator();
    insert_iterator(Container& x, typename Container::iterator i);
    insert_iterator<Container>&
      operator=(const typename Container::value_type& value);
    insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

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

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

24.8.2.6 insert_iterator operations [insert.iter.ops]

24.8.2.6.1 insert_iterator constructor [insert.iter.cons]

```
insert_iterator();
```

1 *Effects:* Value-initializes `container` and `iter`. If `Container::iterator` is a literal type, then this constructor shall be a trivial constructor.

```
insert_iterator(Container& x, typename Container::iterator i);
```

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

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

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

1 *Effects:*

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

2 *Returns:* `*this`.

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

3 *Effects:*

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

4 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

24.8.2.6.4 insert_iterator::operator++ [insert.iter.op++]

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

1 *Returns:* `*this`.

24.8.2.6.5 inserter [inserter]

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

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

24.8.3 Move iterators [move.iterators]

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. ~~Some generic algorithms can be called with move iterators to replace copying with moving.~~ [Editor's note: This is untrue now. The algorithms that do copying are constrained with `IndirectlyCopyable`, which will reject `mov_iterators`.]

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*]

24.8.3.1 Class template move_iterator [move.iterator]

```
namespace std {
  template <~~class Iterator~~WeakInputIterator I>
    requires Same<ReferenceType<I>, ValueType<I>&>
  class move_iterator {
  public:

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

    using iterator_type = I;
    using difference_type = DifferenceType<I>;
    using value_type = ValueType<I>;
    using iterator_category = IteratorCategory<I>;
    using reference = ValueType<I>&&;


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

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

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

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

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

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

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

  template <~~class Iterator1~~WeakInputIterator I1, ~~class Iterator2~~WeakInputIterator I2>
      requires SizedIteratorRange<I2, I1>
    ~~auto~~DifferenceType<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~~DifferenceType<I> n,
        const move_iterator<~~Iterator~~I>& x);
  template <~~class Iterator~~WeakInputIterator I>
    move_iterator<~~Iterator~~I> make_move_iterator(~~Iterator~~I i);
```

```
}
```

24.8.3.2 `move_iterator` requirements [move.iter.requirements]

1 The template parameter `Iterator` shall meet the requirements for an Input Iterator (24.2.14). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.2.18) or a Random Access Iterator (24.2.19), respectively.

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

24.8.3.3.1 `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~~WeakInputIterator U>
  requires Convertible<U, I>
move_iterator(const move_iterator<U>& u);
```

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

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

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

```
template <~~class~~WeakInputIterator U>
  requires Convertible<U, I>
move_iterator& operator=(const move_iterator<U>& u);
```

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

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

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

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

1 *Returns:* `current`.

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

`reference operator*() const;`

1 *Returns:* `std::move(*current)`.

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

`pointer operator->() const;`

1 *Returns:* `current`.

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

`move_iterator& operator++();`

1 *Effects:* `++current`.

2 *Returns:* `*this`.

`move_iterator operator++(int);`

3 *Effects:*

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

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

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

1 *Effects:* `--current`.

2 *Returns:* `*this`.

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

3 *Effects:*

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

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

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

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

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

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

2 *Returns:* `*this`.

24.8.3.3.12 move_iterator::operator[] [move.iter.op.index]

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

1 *Returns:* `std::move(current[n])`.

24.8.3.3.13 move_iterator comparisons [move.iter.op.comp]

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

1 *Returns:* `x.base() == y.base()`.

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

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

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

3 *Returns:* `x.base() < y.base()`.

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

4 *Returns:* `!(y < x)`.

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

5 *Returns:* `y < x`.

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

6 *Returns:* `!(x < y)`.

24.8.3.3.14 move_iterator non-member functions [move.iter.nonmember]

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

1 *Returns:* `x.base() - y.base()`.

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

2 *Returns:* `x + n`.

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

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

24.8.4 Common iterators [common.iterators]

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

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

3 [*Example:*

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

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

— *end example*]

24.8.4.1 Class template `common_iterator` [common.iterator]

```
namespace std {
  // exposition only
  template<class A, class B>
  concept bool __WeaklyEqualityComparable =
    EqualityComparable<A>() && EqualityComparable<B>() &&
    requires(A a, B b) {
      {a==b} -> Boolean;
      {a!=b} -> Boolean;
      {b==a} -> Boolean;
      {b!=a} -> Boolean;
    };
  // exposition only
  template<class I, class S>
  concept bool __WeakSentinel =
    Iterator<I> && Regular<S> &&
    __WeaklyEqualityComparable<I, S>;
```

```
template <InputIterator I, __WeakSentinel<I> S>
  requires !Same<I, S>
class common_iterator {
public:
  using difference_type = DifferenceType<I>;
  using value_type = ValueType<I>;
  using iterator_category =
    conditional_t<ForwardIterator<I>,
                  std::forward_iterator_tag,
                  std::input_iterator_tag>;
  using reference = ReferenceType<I>;

  common_iterator();
  common_iterator(I i);
  common_iterator(S s);
  template <InputIterator U, __WeakSentinel<U> V>
    requires Convertible<U, I> && Convertible<V, S>
  common_iterator(const common_iterator<U, V>& u);
  template <InputIterator U, __WeakSentinel<U> V>
    requires Convertible<U, I> && Convertible<V, S>
  common_iterator& operator=(const common_iterator<U, V>& u);

  ~common_iterator();

  reference operator*() const;

  common_iterator& operator++();
  common_iterator operator++(int);

private:
  bool is_sentinel; // exposition only
  I iter;           // exposition only
  S sent;           // exposition only
};

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

template <InputIterator I1, __WeakSentinel<I1> S1,
          InputIterator I2, __WeakSentinel<I2> S2>
  requires SizedIteratorRange<I1, I1> && SizedIteratorRange<I2, I2> &&
    requires (I1 a, I2 b) { {a-b}->DifferenceType<I2>; {b-a}->DifferenceType<I2>; }
    requires (I1 i, S2 s) { {i-s}->DifferenceType<I2>; {s-i}->DifferenceType<I2>; }
    requires (I2 i, S1 s) { {i-s}->DifferenceType<I2>; {s-i}->DifferenceType<I2>; }
```

```
    DifferenceType<I2> operator-(
      const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
  }
```

1 [*Note:* The use of the expository `@__WeaklyEqualityComparable@` and `@__WeakSentinel@` concepts is avoid the self-referential requirements that would happen if parameters `I` and `S` use `common_iterator<I, S>` as their common type. — *end note*]

2 [*Note:* The ad hoc constraints on `common_iterator`'s `operator-` exist for the same reason. — *end note*]

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

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

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

```
common_iterator();
```

1 *Effects:* Constructs a `common_iterator`, value-initializing `is_sentinel` and `iter`. It is unspecified whether any initialization is performed for `sent`. 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`.

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

2 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `false` and `iter` with `i`. It is unspecified whether any initialization is performed for `sent`.

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

3 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `true` and `sent` with `s`. It is unspecified whether any initialization is performed for `iter`.

```
template <InputIterator U, __WeakSentinel<U> V>
  requires Convertible<U, I> && Convertible<V, S>
common_iterator(const common_iterator<U, V>& u);
```

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

(4.1) — If `u.is_sentinel` is true, `sent` is initialized with `u.sent`. It is unspecified whether any initialization is performed for `iter`.

(4.2) — If `u.is_sentinel` is false, `iter` is initialized with `u.iter`. It is unspecified whether any initialization is performed for `sent`.

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

```
template <InputIterator U, __WeakSentinel<U> V>
  requires Convertible<U, I> && Convertible<V, S>
common_iterator& operator=(const common_iterator<U, V>& u);
```

1 *Effects:* Assigns `u.is_sentinel` to `is_sentinel`.

(1.1) — If `u.is_sentinel` is true, assigns `u.sent` to `sent`. It is unspecified whether any operation is performed on `iter`.

(1.2) — If `u.is_sentinel` is false, assigns `u.iter` to `iter`. It is unspecified whether any operation is performed on `sent`.

2 *Returns:* `*this`

```
~common_iterator();
```

3 *Effects:* Runs the destructor(s) for any members that are currently initialized.

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

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

1 *Requires:* `!is_sentinel`

2 *Returns:* `*iter`.

24.8.4.2.4 `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:*

```
common_iterator tmp = *this;
++iter;
return tmp;
```

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

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

1 *Returns:*

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

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

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

```
template <InputIterator I1, __WeakSentinel<I1> S1,
          InputIterator I2, __WeakSentinel<I2> S2>
  requires SizedIteratorRange<I1, I1> && SizedIteratorRange<I2, I2> &&
    requires (I1 a, I2 b) { {a-b}->DifferenceType<I2>; {b-a}->DifferenceType<I2>; }
```

```
    requires (I1 i, S2 s) { {i-s}->DifferenceType<I2>; {s-i}->DifferenceType<I2>; }
    requires (I2 i, S1 s) { {i-s}->DifferenceType<I2>; {s-i}->DifferenceType<I2>; }
DifferenceType<I2> operator-(
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
```

3 *Returns:*

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

24.8.5 Counted iterators and sentinels [counted.iterators]

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 `counted_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
vector<string> v(make_counted_iterator(s.begin(), 10),
                 counted_sentinel()); // copies 10 strings into v
```

— *end example*]

24.8.5.1 Class template counted_iterator [counted.iterator]

```
namespace std {
  template <WeakInputIterator I>
  class counted_iterator {
  public:
    using iterator_type = I;
    using difference_type = DifferenceType<I>;
    using value_type = ValueType<I>;
    using iterator_category =
      conditional_t<ForwardIterator<I>,
                    IteratorCategory<I>,
                    std::input_iterator_tag>;
    using reference = ReferenceType<I>;

    counted_iterator();
    counted_iterator(I x, DifferenceType<I> n);
    template <WeakInputIterator U>
      requires Convertible<U, I>
    counted_iterator(const counted_iterator<U>& u);
    template <WeakInputIterator U>
      requires Convertible<U, I>
    counted_iterator& operator=(const counted_iterator<U>& u);

    I base() const;
    DifferenceType<I> count() const;
    reference 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>;
    unspecified  operator[](difference_type n) const
      requires RandomAccessIterator<I>;
  protected:
    I current;
    DifferenceType<I> cnt;
  };

  template <WeakInputIterator I1, WeakInputIterator I2>
    bool operator==(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <WeakInputIterator I>
    bool operator==(
      const counted_iterator<I>& x, counted_sentinel y);
  template <WeakInputIterator I>
    bool operator==(
      counted_sentinel x, const counted_iterator<I>& y);
  bool operator==(counted_sentinel x, counted_sentinel y);
  template <WeakInputIterator I1, WeakInputIterator I2>
    bool operator!=(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <WeakInputIterator I>
    bool operator!=(
      const counted_iterator<I>& x, counted_sentinel y);
  template <WeakInputIterator I>
    bool operator!=(
      counted_sentinel x, const counted_iterator<I>& y);
  bool operator!=(counted_sentinel x, counted_sentinel y);

  template <RandomAccessIterator I1, RandomAccessIterator I2>
      requires TotallyOrdered<I1, I2>()
    bool operator<(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <RandomAccessIterator I>
    bool operator<(
      const counted_iterator<I>& x, counted_sentinel y);
  template <RandomAccessIterator I>
    bool operator<(
      counted_sentinel x, const counted_iterator<I>& y);
  bool operator<(counted_sentinel x, counted_sentinel y);
  template <RandomAccessIterator I1, RandomAccessIterator I2>
```

```
      requires TotallyOrdered<I1, I2>()
    bool operator<=(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <RandomAccessIterator I>
    bool operator<=(
      const counted_iterator<I>& x, counted_sentinel y);
  template <RandomAccessIterator I>
    bool operator<=(
      counted_sentinel x, const counted_iterator<I>& y);
  bool operator<=(counted_sentinel x, counted_sentinel y);
  template <RandomAccessIterator I1, RandomAccessIterator I2>
      requires TotallyOrdered<I1, I2>()
    bool operator>(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <RandomAccessIterator I>
    bool operator>(
      const counted_iterator<I>& x, counted_sentinel y);
  template <RandomAccessIterator I>
    bool operator>(
      counted_sentinel x, const counted_iterator<I>& y);
  bool operator>(counted_sentinel x, counted_sentinel y);
  template <RandomAccessIterator I1, RandomAccessIterator I2>
      requires TotallyOrdered<I1, I2>()
    bool operator>=(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <RandomAccessIterator I>
    bool operator>=(
      const counted_iterator<I>& x, counted_sentinel y);
  template <RandomAccessIterator I>
    bool operator>=(
      counted_sentinel x, const counted_iterator<I>& y);
  bool operator>=(counted_sentinel x, counted_sentinel y);

  template <WeakInputIterator I1, WeakInputIterator I2>
    DifferenceType<I2> operator-(
      const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  template <WeakInputIterator I>
    DifferenceType<I> operator-(
      const counted_iterator<I>& x, counted_sentinel y);
  template <WeakInputIterator I>
    DifferenceType<I> operator-(
      counted_sentinel x, const counted_iterator<I>& y);
  ptrdiff_t operator-(counted_sentinel x, counted_sentinel y);
  template <RandomAccessIterator I>
    counted_iterator<I>
      operator+(DifferenceType<I> n, const counted_iterator<I>& x);
  template <WeakInputIterator I>
    counted_iterator<I> make_counted_iterator(I i, DifferenceType<I> n);

  template <WeakInputIterator I>
    void advance(counted_iterator<I>& i, DifferenceType<I> n);
}
```

24.8.5.2 counted_iterator operations [counted.iter.ops]

24.8.5.2.1 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, DifferenceType<I> n);
```

2 *Requires:* `n >= 0`

3 *Effects:* Constructs a `counted_iterator`, initializing `current` with `i` and `cnt` with `n`.

```
template <WeakInputIterator U>
  requires Convertible<U, I>
count_iterator(const counted_iterator<U>& u);
```

4 *Effects:* Constructs a `counted_iterator`, initializing `current` with `u.base()` and `cnt` with `u.count()`.

24.8.5.2.2 `counted_iterator::operator=` [counted.iter.op=]

```
template <WeakInputIterator U>
  requires Convertible<U, I>
counted_iterator& operator=(const counted_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to `current` and `u.count()` to `cnt`.

24.8.5.2.3 `counted_iterator` conversion [counted.iter.op.conv]

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

1 *Returns:* `current`.

24.8.5.2.4 `counted_iterator` count [counted.iter.op.cnt]

```
DifferenceType<I> count() const;
```

1 *Returns:* `cnt`.

24.8.5.2.5 `count_iterator::operator*` [counted.iter.op.star]

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

1 *Returns:* `*current`.

24.8.5.2.6 `counted_iterator::operator++` [counted.iter.op.incr]

```
counted_iterator& operator++();
```

1 *Requires:* `cnt > 0`

2 *Effects:*

```
++current;
--cnt;
```

3 *Returns:* `*this`.

```
counted_iterator operator++(int);
```

4 *Requires:* `cnt > 0`

5 *Effects:*

```
counted_iterator tmp = *this;
++current;
--cnt;
return tmp;
```

24.8.5.2.7 `counted_iterator::operator--` [counted.iter.op.decr]

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

1 *Effects:*

```
--current;
++cnt;
```

2 *Returns:* `*this`.

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

3 *Effects:*

```
counted_iterator tmp = *this;
--current;
++cnt;
return tmp;
```

24.8.5.2.8 `counted_iterator::operator+` [counted.iter.op.+]

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

1 *Requires:* `n <= cnt`

2 *Returns:* `counted_iterator(current + n, cnt - n)`.

24.8.5.2.9 `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`.

24.8.5.2.10 `counted_iterator::operator-` [counted.iter.op.-]

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

1 *Requires:* `-n <= cnt`

2 *Returns:* `counted_iterator(current - n, cnt + n)`.

24.8.5.2.11 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`.

24.8.5.2.12 counted_iterator::operator[] [counted.iter.op.index]

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

1 *Requires:* `n <= cnt`

2 *Returns:* `current[n]`.

24.8.5.2.13 counted_iterator comparisons [counted.iter.op.comp]

```
template <WeakInputIterator I1, WeakInputIterator I2>
  bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Returns:* `x.base() == y.base()` if `EqualityComparable<I1, I2>()`; otherwise, `x.count() == y.count()`.

```
template <WeakInputIterator I>
  bool operator==(
    const counted_iterator<I>& x, counted_sentinel y);
```

2 *Returns:* `x.count() == 0`.

```
template <WeakInputIterator I>
  bool operator==(
    counted_sentinel x, const counted_iterator<I>& y);
```

3 *Returns:* `y.count() == 0`.

```
bool operator==(counted_sentinel x, counted_sentinel y);
```

4 *Returns:* `true`.

```
template <WeakInputIterator I1, WeakInputIterator I2>
  bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <WeakInputIterator I>
  bool operator!=(
    const counted_iterator<I>& x, counted_sentinel y);
template <WeakInputIterator I>
  bool operator!=(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator!=(counted_sentinel x, counted_sentinel y);
```

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

```
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

6 *Returns:* `x.base() < y.base()`.

```
template <RandomAccessIterator I>
  bool operator<(
    const counted_iterator<I>& x, counted_sentinel y);
```

7 *Returns:* `x.count() != 0`.

```
template <RandomAccessIterator I>
  bool operator<(
    counted_sentinel x, const counted_iterator<I>& y);
```

8 *Returns:* `false`.

```
bool operator<(counted_sentinel x, counted_sentinel y);
```

9 *Returns:* `false`.

```
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator<=(
    const counted_iterator<I>& x, counted_sentinel y);
template <RandomAccessIterator I>
  bool operator<=(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator<=(counted_sentinel x, counted_sentinel y);
```

10 *Returns:* `!(y < x)`.

```
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator>(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator>(
    const counted_iterator<I>& x, counted_sentinel y);
template <RandomAccessIterator I>
  bool operator>(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator>(counted_sentinel x, counted_sentinel y);
```

11 *Returns:* `y < x`.

```
template <RandomAccessIterator I1, RandomAccessIterator I2>
    requires TotallyOrdered<I1, I2>()
  bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
template <RandomAccessIterator I>
  bool operator>=(
    const counted_iterator<I>& x, counted_sentinel y);
```

```
template <RandomAccessIterator I>
  bool operator>=(
    counted_sentinel x, const counted_iterator<I>& y);
bool operator>=(counted_sentinel x, counted_sentinel y);
```

12 *Returns:* `!(x < y)`.

24.8.5.2.14 counted_iterator non-member functions [counted.iter.nonmember]

```
template <WeakInputIterator I1, WeakInputIterator I2>
  DifferenceType<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Returns:* `x.base() - y.base()` if `SizedIteratorRange<I2, I1>`; otherwise, `y.count() - x.count()`.

```
template <WeakInputIterator I>
  DifferenceType<I> operator-(
    const counted_iterator<I>& x, counted_sentinel y);
```

2 *Returns:* `-x.count()`.

```
template <WeakInputIterator I>
  DifferenceType<I> operator-(
    counted_sentinel x, const counted_iterator<I>& y);
```

3 *Returns:* `y.count()`.

```
ptrdiff_t operator-(counted_sentinel x, counted_sentinel y);
```

4 *Returns:* `0`.

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

5 *Requires:* `n <= x.count()`.

6 *Returns:* `x + n`.

```
template <WeakInputIterator I>
  counted_iterator<I> make_counted_iterator(I i, DifferenceType<I> n);
```

7 *Requires:* `n >= 0`.

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

```
template <WeakInputIterator I>
  void advance(counted_iterator<I>& i, DifferenceType<I> n);
```

9 *Requires:* `n <= i.count()`.

10 *Effects:*

```
i = make_counted_iterator(next(i.base(), n), i.count() - n);
```

24.8.5.3 Counted sentinel [counted.sentinel]

1 Class `counted_sentinel` is an empty type used to represent the end of a counted range. It is used together with class template `counted_iterator`(24.8.5.1) to denote a range of elements that starts at a known position and includes the subsequent N elements.

```
namespace std {
  class counted_sentinel { };
}
```

24.8.5.4 Specializations of `common_type` [counted.traits.specializations]

```
namespace std {
  template<WeakInputIterator I>
  struct common_type<counted_iterator<I>, counted_sentinel> {
    using type = common_iterator<counted_iterator<I>, counted_sentinel>;
  };
  template<WeakInputIterator I>
  struct common_type<counted_sentinel, counted_iterator<I>> {
    using type = common_iterator<counted_iterator<I>, counted_sentinel>;
  };
}
```

1 [*Note:* By specializing `common_type` this way, `counted_iterator` and `counted_sentinel` can satisfy the `Common` requirement of the `EqualityComparable` concept. — *end note*]

24.8.6 Unreachable sentinel [unreachable.sentinels]

24.8.6.1 Class `unreachable sentinel` [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 make the call to `find` more efficient since the loop test against the sentinel does not require a conditional branch. — *end example*]

```
namespace std {
  class unreachable { };
  template <Iterator I>
    constexpr bool operator==(I const &, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator==(unreachable, I const &) noexcept;
  constexpr bool operator==(unreachable, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator!=(I const &, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator!=(unreachable, I const &) noexcept;
  constexpr bool operator!=(unreachable, unreachable) noexcept;
}
```

24.8.6.2 `unreachable` operations [unreachable.sentinel.ops]

24.8.6.2.1 `operator==` [unreachable.sentinel.op==]

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

1 *Returns:* `false`.

```
constexpr bool operator==(unreachable, unreachable) noexcept;
```

2 *Returns:* `true`.

24.8.6.2.2 `operator!=` [unreachable.sentinel.op!=]

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

1 *Returns:* `!(x == y)`

24.8.6.3 Specializations of `common_type` [unreachable.traits.specializations]

```
namespace std {
  template<Iterator I>
  struct common_type<I, unreachable> {
    using type = common_iterator<I, unreachable>;
  };
  template<Iterator I>
  struct common_type<unreachable, I> {
    using type = common_iterator<I, unreachable>;
  };
}
```

1 [*Note:* By specializing `common_type` this way, any iterator and `unreachable` can satisfy the `Common` requirement of the `EqualityComparable` concept. — *end note*]

24.9 Stream iterators [stream.iterators]

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*]

24.9.1 Class template `istream_iterator` [istream.iterator]

1 The class template `istream_iterator` is an input iterator (24.2.14) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. ~~The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned.~~ The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

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 {
  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 charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT,traits> istream_type;
    see below istream_iterator();
    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!=(const istream_iterator<T,charT,traits,Distance>& x,
            const istream_iterator<T,charT,traits,Distance>& y);
}
```

24.9.1.1 istream_iterator constructors and destructor [istream.iterator.cons]

`see below istream_iterator();`

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then this constructor shall be a `constexpr` constructor.

2 *Postcondition:* `in_stream == 0`.

`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.

24.9.1.2 istream_iterator operations [istream.iterator.ops]

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

1 *Returns:* *value*.

```
const T* operator->() const;
```

2 *Returns:* `&(operator*())`.

```
istream_iterator<T,charT,traits,Distance>& operator++();
```

3 *Requires:* `in_stream != 0`.

4 *Effects:* `*in_stream >>value`.

5 *Returns:* `*this`.

```
istream_iterator<T,charT,traits,Distance> operator++(int);
```

6 *Requires:* `in_stream != 0`.

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

9 *Returns:* `!(x == y)`

24.9.2 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 {
  template <class T, class charT = char, class traits = char_traits<charT> >
  class ostream_iterator:
    public iterator<output_iterator_tag, void, void, void, void> {
  public:
    typedef output_iterator_tag iterator_category;
    typedef ptrdiff_t difference_type;
    typedef charT char_type;
    typedef traits traits_type;
```

```
    typedef basic_ostream<charT,traits> ostream_type;
    constexpr ostream_iterator() noexcept;
    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const charT* delimiter);
    ostream_iterator(const ostream_iterator<T,charT,traits>& x);
   ~ostream_iterator();
    ostream_iterator<T,charT,traits>& operator=(const T& value);

    ostream_iterator<T,charT,traits>& operator*();
    ostream_iterator<T,charT,traits>& operator++();
    ostream_iterator<T,charT,traits>& operator++(int);
  private:
    basic_ostream<charT,traits>* out_stream;  // exposition only
    const charT* delim;                       // exposition only
  };
}
```

24.9.2.1 ostream_iterator constructors and destructor [ostream.iterator.cons.des]

`constexpr ostream_iterator() noexcept;`

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

`ostream_iterator(ostream_type& s);`

2 *Effects:* Initializes *out_stream* with `&s` and *delim* with null.

`ostream_iterator(ostream_type& s, const charT* delimiter);`

3 *Effects:* Initializes *out_stream* with `&s` and *delim* with `delimiter`.

`ostream_iterator(const ostream_iterator& x);`

4 *Effects:* Constructs a copy of `x`.

`~ostream_iterator();`

5 *Effects:* The iterator is destroyed.

24.9.2.2 ostream_iterator operations [ostream.iterator.ops]

`ostream_iterator& operator=(const T& value);`

1 *Effects:*

```
*out_stream << value;
if(delim != 0)
  *out_stream << delim;
return (*this);
```

2 *Requires:* `out_stream!= 0`.

`ostream_iterator& operator*();`

3 *Returns:* `*this`.

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

4 *Returns:* `*this`.

24.9.3 Class template `istreambuf_iterator` [istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (24.2.14) that reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. ~~[*Note:* operator-> may return a proxy. — *end note*]~~ Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the *end-of-stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` 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 {
  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 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;
    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!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
}
```

24.9.3.1 Class template `istreambuf_iterator::proxy` [istreambuf.iterator::proxy]

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

1 Class `istreambuf_iterator<charT,traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

24.9.3.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

`constexpr istreambuf_iterator() noexcept;`

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

```
istreambuf_iterator(basic_istream<charT,traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT,traits>* s) noexcept;
```

2 *Effects:* Constructs an `istreambuf_iterator<>` that uses the `basic_streambuf<>` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

`istreambuf_iterator(const proxy& p) noexcept;`

3 *Effects:* Constructs a `istreambuf_iterator<>` that uses the `basic_streambuf<>` object pointed to by the `proxy` object's constructor argument `p`.

24.9.3.3 `istreambuf_iterator::operator*` [istreambuf.iterator::op*]

`charT operator*() const`

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

24.9.3.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```
istreambuf_iterator<charT,traits>&
    istreambuf_iterator<charT,traits>::operator++();
```

1 *Effects:* `sbuf_->sbumpc()`.

2 *Returns:* `*this`.

`proxy istreambuf_iterator<charT,traits>::operator++(int);`

3 *Returns:* `proxy(sbuf_->sbumpc(), sbuf_)`.

24.9.3.5 `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.

24.9.3.6 `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:* `a.equal(b)`.

24.9.3.7 `operator!=` [istreambuf.iterator::op!=]

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

1 *Returns:* `!a.equal(b)`.

24.9.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```
namespace std {
  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.

24.9.4.1 `ostreambuf_iterator` constructors [ostreambuf.iter.cons]

```
constexpr ostreambuf_iterator() noexcept;
```

1 *Effects:* Initializes `sbuf_` with null.

```
ostreambuf_iterator(ostream_type& s) noexcept;
```

2 *Requires:* `s.rdbuf()` shall not null pointer.

3 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

```
ostreambuf_iterator(streambuf_type* s) noexcept;
```

4 *Requires:* `s` shall not be a null pointer.

5 *Effects:* Initializes `sbuf_` with `s`.

24.9.4.2 `ostreambuf_iterator` operations [ostreambuf.iter.ops]

```
ostreambuf_iterator<charT,traits>&
  operator=(charT c);
```

1 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

2 *Requires:* `sbuf_ != 0`.

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

7 *Requires:* `sbuf_ != 0`.

24.10 Iterable concepts [iterables]

24.10.1 General [iterables.general]

1 This subclause describes components for dealing with ranges of elements.

2 The following subclauses describe iterable and range requirements, and components for iterable primitives, predefined ranges, and stream ranges, as summarized in Table 4.

Table 4 — Iterables library summary

	Subclause	Header(s)
24.10.2	Requirements	
??	Iterable primitives	`<iterator>`
??	Predefined ranges	
??	Stream ranges	

24.10.2 Iterable requirements [iterables.requirements]

24.10.2.1 In general [iterables.requirements.general]

1 Iterables are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. It their simplest form, an iterable object is one on which one can call `begin` and `end` to get an iterator (24.2.11) and a sentinel (24.2.12) or an iterator. To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterables.

2 This International Standard defines three fundamental categories of iterables based on the syntax and semantics supported by each: *iterable*, *sized iterable* and *range*, as shown in Table 5.

Table 5 — Relations among iterable categories

$$\begin{tikzcd} \textbf{Sized Iterable} \arrow[dr] & \\ & \textbf{Iterable} \\ \textbf{Range} \arrow[ur] & \end{tikzcd}$$

3 The `Iterable` concept requires only that `begin` and `end` return an iterator and a sentinel. [*Note:* An iterator is a valid sentinel. — *end note*] The `SizedIterable` concept refines `Iterable` but adds the requirement that the number of elements in the iterable can be determined in constant time with the `size` function. The `Range` concept describes requirements on an `Iterable` type with constant-time copy and assign operations.

4 In addition to the three fundamental iterable categories, this standard defines a number of convenience refinements of `Iterable` that group together requirements that appear often in the concepts, algorithms, and range views. *Bounded iterables* are iterables for which `begin` and `end` return objects of the same type. *Random access iterables* are iterables for which `begin` returns a model of `RandomAccessIterator` (24.2.19). The iterable categories *bidirectional iterable*, *forward iterable*, *input iterable* and *output iterable* are defined similarly. [*Note:* There is no *weak input iterable* or *weak output iterable* because of the `EqualityComparable` requirement on iterators and sentinels. — *end note*] [Editor's note: TODO: Rethink that because a weak input iterable would not require (strongly) incrementable iterators.]

24.10.2.2 Iterables [iterable.iterables]

1 The `Iterable` concept defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and an `end` iterator or sentinel.

```
template <class T>
concept bool Iterable =
  requires(T t) {
    typename IteratorType<T>;
    typename SentinelType<T>;
    { begin(t) } -> IteratorType<T>;
    { end(t) } -> SentinelType<T>;
    requires Sentinel<SentinelType<T>, IteratorType<T>>;
  };
```

`begin` and `end` are required to be amortized constant time. [*Note:* Most algorithms requiring this concept simply forward to an `Iterator`-based algorithm by calling `begin` and `end`. — *end note*]

24.10.2.3 Sized iterables [sized.iterables]

1 The `SizedIterable` concept describes the requirements of an `Iterable` type that knows its size in constant time with the `size` function.

```
// exposition only
template <class T>
concept bool __SizedIterableLike =
  Iterable<T> &&
  requires(T t) {
    { size(t) } -> Integral;
    requires Convertible<decltype(size(t)),
                         DifferenceType<IteratorType<T>>>;
  };

template <class T>
```

```
concept bool SizedIterable =
  __SizedIterableLike<T> && is_sized_iterable<T>::value;
```

2 For any type `T`, `is_sized_iterable<T>` derives from `true_type` if `T` models `__SizedIterableLike`, and `false_type` otherwise.

3 Users are free to specialize `is_sized_iterable`. [*Note:* Users may specialize `is_sized_iterable` to override the default in the case of accidental conformance. — *end note*]

4 [*Note:* A possible implementation for `is_sized_iterable` is given below:

```
template <class R>
struct is_sized_iterable : false_type { };

template <__SizedIterableLike R>
struct is_sized_iterable<R> : true_type { };
```

— *end note*]

24.10.2.4 Ranges [range.iterables]

1 The `Range` concept describes the requirements of an `Iterable` 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 Range.

2 [*Example:* Examples of `Range`s are:

(2.1) — An `Iterable` type that wraps a pair of iterators.

(2.2) — An `Iterable` type that hold its elements by `shared_ptr` and shares ownership with all its copies.

(2.3) — An `Iterable` type that generates its elements on demand.

A container (23) is not a `Range` since copying the container copies the elements, which cannot be done in constant time. — *end example*]

```
template <class T>
concept bool Range =
  Iterable<T> && Semiregular<T> && is_range<T>::value;
```

3 Since the difference between `Iterable` and `Range` is largely semantic, the two are differentiated with the help of the `is_range` trait. Users may specialize the `is_range` trait. By default, `is_range` uses the following heuristics to determine whether an `Iterable` type `T` is a `Range`:

(3.1) — If `T` derives from `range_base`, `is_range<T>::value` is true.

(3.2) — If a top-level `const` changes `T`'s `IteratorType`'s `ReferenceType` type, `is_range<T>::value` is false. [*Note:* Deep `const`-ness implies element ownership, whereas shallow `const`-ness implies reference semantics. — *end note*]

4 [*Note:* Below is a possible implementation of the `is_range` trait.

```
struct range_base { };

// exposition only
template <class T>
concept bool __ContainerLike =
  Iterable<T> &&
  !Same<ReferenceType<IteratorType<T &>>,
        ReferenceType<IteratorType<const T &>>>;
```

```
// exposition only
template <class T>
struct is_range : false_type { };

template <Iterable T>
  requires Derived<remove_reference_t<T>, range_base> ||
    !__ContainerLike<remove_reference_t<T>>
struct is_range<T> : true_type { };
```

[Editor's note: REVIEW: Will this create ambiguities with user specializations?] — *end note*]

24.10.2.5 Bounded iterables [bounded.iterables]

1 The `BoundedIterable` concept describes requirements of an `Iterable` type for which `begin` and `end` return objects of the same type. [*Note:* The standard containers (23) are models of `BoundedIterable`. — *end note*]

```
template <class T>
concept bool BoundedIterable =
  Iterable<T> && Same<IteratorType<T>, SentinelType<T>>;
```

24.10.2.6 Input iterables [input.iterables]

1 The `InputIterable` concept describes requirements of an `Iterable` type for which `begin` returns a model of `InputIterator` (24.2.14).

```
template <class T>
concept bool InputIterable =
  Iterable<T> && InputIterator<IteratorType<T>>;
```

24.10.2.7 Forward iterables [forward.iterables]

1 The `ForwardIterable` concept describes requirements of an `InputIterable` type for which `begin` returns a model of `ForwardIterator` (24.2.17).

```
template <class T>
concept bool ForwardIterable =
  InputIterable<T> && ForwardIterator<IteratorType<T>>;
```

24.10.2.8 Bidirectional iterables [bidirectional.iterables]

1 The `BidirectionalIterable` concept describes requirements of a `ForwardIterable` type for which `begin` returns a model of `BidirectionalIterator` (24.2.18).

```
template <class T>
concept bool BidirectionalIterable =
  ForwardIterable<T> && BidirectionalIterator<IteratorType<T>>;
```

24.10.2.9 Random access iterables [random.access.iterables]

1 The `RandomAccessIterable` concept describes requirements of a `BidirectionalIterable` type for which `begin` returns a model of `RandomAccessIterator` (24.2.19).

```
template <class T>
concept bool RandomAccessIterable =
  BidirectionalIterable<T> && RandomAccessIterator<IteratorType<T>>;
```

24.11 range access [iterator.range]

1 In addition to being available via inclusion of the `<iterator>` header, the function templates in 24.11 are available when any of the following headers are included: `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<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)`.

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 6.

Table 6 — Algorithms library summary

	Subclause	Header(s)
25.2	Non-modifying sequence operations	
25.3	Mutating sequence operations	`<algorithm>`
25.4	Sorting and related operations	
25.5	C library algorithms	`<cstdlib>`

[Editor's note: Remove the header `<algorithm>` synopsis.]

Header `<algorithm>` synopsis

```
#include <initializer_list>

namespace std {

  // 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<InputIterable Rng, class Proj = identity,
      IndirectCallablePredicate<Projected<IteratorType<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<InputIterable Rng, class Proj = identity,
      IndirectCallablePredicate<Projected<IteratorType<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<InputIterable Rng, class Proj = identity,
      IndirectCallablePredicate<Projected<IteratorType<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>
  pair<I, Fun>
    for_each(I first, S last, Fun f, Proj proj = Proj{});

template<InputIterable Rng, class Proj = identity,
    IndirectCallable<Projected<IteratorType<Rng>, Proj>> Fun>
  pair<IteratorType<Rng>, Fun>
    for_each(Rng& rng, Fun f, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  I find(I first, S last, const T& value, Proj proj = Proj{});

template<InputIterable Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  IteratorType<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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  IteratorType<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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  IteratorType<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<ForwardIterable Rng1, ForwardIterable Rng2, class Proj = identity,
    IndirectCallableRelation<IteratorType<Rng2>,
      Projected<IteratorType<Rng>, Proj>> Pred = equal_to<>>
  IteratorType<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<InputIterable Rng1, ForwardIterable Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Pred = equal_to<>>
  IteratorType<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<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Pred = equal_to<>>
  IteratorType<Rng>
    adjacent_find(Rng& rng, Pred pred = Pred{}, Proj proj = Proj{});

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

template<InputIterable Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  DifferenceType<IteratorType<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  DifferenceType<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  DifferenceType<IteratorType<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallablePredicate<Projected<I1, Proj1>, Projected<I2, Proj2>> Pred = equal_to<>>
  pair<I1, I2>
    mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, WeakInputIterator I2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng1>, Proj1>,
      Projected<I2, Proj2>> Pred = equal_to<>>
  pair<IteratorType<Rng1>, 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<>>
    pair<I1, I2>
      mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<InputIterable Rng1, InputIterable Rng2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectCallablePredicate<Projected<IteratorType<Rng1>, Proj1>,
        Projected<IteratorType<Rng2>, Proj2>> Pred = equal_to<>>
    pair<IteratorType<Rng1>, IteratorType<Rng2>>
      mismatch(Rng1& rng1, Rng2& rng2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
      class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
    bool equal(I1 first1, S1 last1,
               I2 first2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<InputIterable Rng1, WeakInputIterator I2, class Pred = equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<IteratorType<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<InputIterable Rng1, InputIterable Rng2, class Pred = equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<IteratorType<Rng1>, IteratorType<Rng2>, Pred, Proj1, Proj2>
    bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
               Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
      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<ForwardIterable Rng1, ForwardIterator I2, class Pred = equal_to<>,
      class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<IteratorType<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<ForwardIterable Rng1, ForwardIterable Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, IteratorType<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<ForwardIterable Rng1, ForwardIterable Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, IteratorType<Rng2>, Pred, Proj1, Proj2>
  IteratorType<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<I1, const T*, Pred, Proj>
  I
    search_n(I first, S last, DifferenceType<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template<ForwardIterable Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, const T*, Pred, Proj>
  IteratorType<Rng>
    search_n(Rng& rng, DifferenceType<IteratorType<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>
  pair<I, O>
    copy(I first, S last, O result);

template<InputIterable Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
```

```
  pair<IteratorType<Rng>, O>
    copy(Rng& rng, O result);

template<WeakInputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  pair<I, O>
    copy_n(I first, iterator_distance_t<I> n, O result);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  pair<I, O>
    copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    copy_if(Rng& rng, O result, Pred pred, Proj proj = Proj{});

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>
  pair<I1, I2>
    copy_backward(I1 first, I1 last, I2 result);

template<BidirectionalIterable Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<IteratorType<Rng>, I>
  pair<IteratorType<Rng>, I>
    copy_backward(Rng& rng, I result);

// 25.3.2, move:
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>
  pair<I, O>
    move(I first, S last, O result);

template<InputIterable Rng, WeaklyIncrementable O>
  requires IndirectlyMovable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    move(Rng& rng, O result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>
  pair<I1, I2>
    move_backward(I1 first, I1 last, I2 result);

template<BidirectionalIterable Rng, BidirectionalIterator I>
  requires IndirectlyMovable<IteratorType<Rng>, I>
  pair<IteratorType<Rng>, I>
    move_backward(Rng& rng, I result);

// 25.3.3, swap:
template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
  requires IndirectlySwappable<I1, I2>
  pair<I1, I2>
```

```
      swap_ranges(I1 first1, S1 last1, I2 first2);

  template<ForwardIterable Rng, ForwardIterator I>
    requires IndirectlySwappable<IteratorType<Rng>, I>
    pair<IteratorType<Rng>, I>
      swap_ranges(Rng& rng1, I first2);

  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
    requires IndirectlySwappable<I1, I2>
    pair<I1, I2>
      swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

  template<ForwardIterable Rng1, ForwardIterable Rng2>
    requires IndirectlySwappable<IteratorType<Rng1>, IteratorType<Rng2>>
    pair<IteratorType<Rng1>, IteratorType<Rng2>>
      swap_ranges(Rng1& rng1, Rng2& rng2);

  template<InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectCallable<Projected<I, Proj>> F,
      WeakOutputIterator<IndirectCallableResultType<F, Projected<I, Proj>>> O>
    pair<I, O>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

  template<InputIterable Rng, class Proj = identity,
      IndirectCallable<Projected<IteratorType<Rng>, Proj>> F,
      WeakOutputIterator<IndirectCallableResultType<F,
        Projected<IteratorType<Rng>, Proj>>> O>
    pair<IteratorType<Rng>, O>
      transform(Rng& rng, O result, F op, Proj proj = Proj{});

  template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectCallable<Projected<I1, Proj1>, Projected<I2, Proj2>> F,
      WeakOutputIterator<IndirectCallableResultType<F, Projected<I1, Proj1>,
        Projected<I2, Proj2>>> O>
  tuple<I1, I2, O>
      transform(I1 first1, S1 last1, I2 first2, O result,
                F binary_op, Proj1 proj1 = Proj1{}, proj2 proj2 = Proj2{});

  template<InputIterable Rng, WeakInputIterator I,
      class Proj1 = identity, class Proj2 = identity,
      IndirectCallable<Projected<IteratorType<Rng>, Proj1>, Projected<I, Proj2>> F,
      WeakOutputIterator<IndirectCallableResultType<F,
        Projected<IteratorType<Rng>, Proj1>, Projected<I, Proj2>>> O>
    tuple<IteratorType<Rng>, I, 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,
      class Proj1 = identity, class Proj2 = identity,
      IndirectCallable<Projected<I1, Proj1>, Projected<I2, Proj2>> F,
      WeakOutputIterator<IndirectCallableResultType<F, Projected<I1, Proj1>,
        Projected<I2, Proj2>>> O>
    tuple<I1, I2, O>
    transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
```

```
          F binary_op, Proj1 proj1 = Proj1{}, proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallable<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> F,
    WeakOutputIterator<IndirectCallableResultType<F,
      Projected<IteratorType<Rng1>, Proj1>, Projected<IteratorType<Rng2>, Proj2>>> O>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, O>
    transform(Rng1& rng1, Rng2& rng2, O result,
              F binary_op, Proj1 proj1 = Proj1{}, proj2 proj2 = Proj2{});
```

[Editor's note: REVIEW: In the Palo Alto proposal, `replace` requires only InputIterators. In C++14, it requires Forward.]

```
template<ForwardIterator I, Sentinel<I> S, class T1, Semiregular T2, class Proj = identity>
  requires Writable<I, T2> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T1 *>
  I
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template<ForwardIterable Rng, class T1, Semiregular T2, class Proj = identity>
  requires Writable<IteratorType<Rng>, T2> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T1 *>
  IteratorType<Rng>
    replace(Rng& rng, const T1& old_value, const T2& new_value);

template<ForwardIterator I, Sentinel<I> S, Semiregular T, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires Writable<I, T>
  I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template<ForwardIterable Rng, Semiregular T, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Writable<IteratorType<Rng>, T>
  IteratorType<Rng>
    replace_if(Rng& rng, Pred pred, const T& new_value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class T1, Semiregular T2, WeakOutputIterator<T2> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T1 *>
  pair<I, O>
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template<InputIterable Rng, class T1, Semiregular T2, WeakOutputIterator<T2> O,
    class Proj = identity>
  requires IndirectlyCopyable<IteratorType<Rng>, O> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T1 *>
  pair<IteratorType<Rng>, O>
    replace_copy(Rng& rng, O result, const T1& old_value, const T2& new_value,
```

```
                 Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, Semiregular T, WeakOutputIterator<T> O,
    class Proj = identity, IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  pair<I, O>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template<InputIterable Rng, Semiregular T, WeakOutputIterator<T> O, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    replace_copy_if(Rng& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});
```

[Editor's note: REVIEW: N3351 only requires WeaklyIncrementable for fill and generate]

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

template<Semiregular T, OutputIterable<T> Rng>
  IteratorType<Rng>
    fill(Rng& rng, const T& value);

template<Semiregular T, WeakOutputIterator<T> O>
  O fill_n(O first, DifferenceType<O> n, const T& value);

template<Function F, OutputIterator<ResultType<F>> O,
    Sentinel<O> S>
  O generate(O first, S last, F gen);

template<Function F, OutputIterable<ResultType<F>> Rng>
  IteratorType<Rng>
    generate(Rng& rng, F gen);

template<Function F, WeakOutputIterator<ResultType<F>> O>
  O generate_n(O first, DistanceType<O> n, F gen);

template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  I remove(I first, S last, const T& value, Proj proj = Proj{});

template<ForwardIterable Rng, class T, class Proj = identity>
  requires Permutable<IteratorType<Rng>> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  IteratorType<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<ForwardIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Permutable<IteratorType<Rng>>
  IteratorType<Rng>
    remove_if(Rng& rng, Pred pred, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  pair<I, O> remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<IteratorType<Rng>, O> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  pair<IteratorType<Rng>, 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>
  pair<I, O>
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> R = equal_to<>>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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>
  pair<I, O>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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<BidirectionalIterable Rng>
  requires Permutable<IteratorType<Rng>>
  IteratorType<Rng>
    reverse(Rng& rng);

template<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  pair<I, O> reverse_copy(I first, S last, O result);

template<BidirectionalIterable Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    reverse_copy(Rng& rng, O result);
```

[Editor's note: Could return a `range` instead of a `pair`. See Future Work annex (C.4).]

```
template<ForwardIterator I, Sentinel<I> S>
  requires Permutable<I>
  pair<I, I> rotate(I first, I middle, S last);

template<ForwardIterable Rng>
  requires Permutable<IteratorType<Rng>>
  pair<IteratorType<Rng>, IteratorType<Rng>>
    rotate(Rng& rng, IteratorType<Rng> middle);

template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  pair<I, O>
    rotate_copy(I first, I middle, S last, O result);

template<ForwardIterable Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    rotate_copy(Rng& rng, IteratorType<Rng> middle, O result);

// 25.3.12, shuffle:
template<RandomAccessIterator I, Sentinel<I> S, UniformRandomNumberGenerator Gen>
  requires Permutable<I> && Convertible<ResultType<Gen>, DifferenceType<I>>
  I shuffle(I first, S last, Gen&& g);

template<RandomAccessIterable Rng, UniformRandomNumberGenerator Gen>
  requires Permutable<I> && Convertible<ResultType<Gen>, DifferenceType<I>>
  IteratorType<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<InputIterable Rng, class Proj = identity,
```

```
    IndirectCallablePredicate<Projected<IteratorType<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<ForwardIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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<BidirectionalIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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>
  tuple<I, O1, O2>
    partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O1> &&
    IndirectlyCopyable<IteratorType<Rng>, O2>
  tuple<IteratorType<Rng>, O1, O2>
    partition_copy(Rng& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
```

[Editor's note: A new algorithm, needed by stable_partition.]

```
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires IndirectlyMovable<I, O1> && IndirectlyMovable<I, O2>
  tuple<I, O1, O2>
    partition_move(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
```

```
  requires IndirectlyMovable<IteratorType<Rng>, O1> &&
    IndirectlyMovable<IteratorType<Rng>, O2>
  tuple<IteratorType<Rng>, O1, O2>
    partition_move(Rng& rng, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<Rng>
    partial_sort(Rng& rng, IteratorType<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});

template<InputIterator I1, Sentinel<I> S1, RandomAccessIterator I2, Sentinel<I> S2,
    class R = less<>, class Proj = identity>
  requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj>
  I2
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                      Comp comp = Comp{}, Proj proj = Proj{});
```

```
template<InputIterable Rng1, RandomAccessIterable Rng2, class R = less<>,
    class Proj = identity>
  requires IndirectlyCopyable<IteratorType<Rng1>, IteratorType<Rng2>> &&
      Sortable<IteratorType<Rng2>, Comp, Proj>
  IteratorType<Rng2>
    partial_sort_copy(Rng1& rng, Rng2& result_rng, Comp comp = Comp{},
                      Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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, Proj proj = Proj{});

template<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<Rng>
    nth_element(Rng& rng, IteratorType<Rng> nth, Comp comp, Proj proj = Proj{});

// 25.4.3, binary search:
template<ForwardIterator I, Sentinel<I> S, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  I
    lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<Rng>
    lower_bound(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  I
    upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
```

```
  IteratorType<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.4).]

```
template<ForwardIterator I, Sentinel<I> S, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  pair<I, I>
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  pair<IteratorType<Rng>, IteratorType<Rng>>
    equal_range(Rng& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterator I, Sentinel<I> S, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  bool
    binary_search(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

// 25.4.4, merge:
```

[Editor's note: REVIEW: Why does the Palo Alto TR require `Incrementable` instead of `WeaklyIncrementable`?]

```
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    Incrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tuple<I1, I2, O>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, Incrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, O>
    merge(Rng1& rng1, Rng2& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

[Editor's note: A new algorithm, needed by `inplace_merge` and `stable_sort`.]

```
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    Incrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
```

```
    requires MergeMovable<I1, I2, O, Comp, Proj1, Proj2>
    tuple<I1, I2, O>
      merge_move(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<InputIterable Rng1, InputIterable Rng2, Incrementable O, class Comp = less<>,
      class Proj1 = identity, class Proj2 = identity>
    requires MergeMovable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
    tuple<IteratorType<Rng1>, IteratorType<Rng2>, O>
      merge_move(Rng1& rng1, Rng2& rng2, O result,
                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    I
      inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template<BidirectionalIterable Rng, class Comp = less<>, class Proj = identity>
    requires Sortable<IteratorType<Rng>, Comp, Proj>
    IteratorType<Rng>
      inplace_merge(Rng& rng, IteratorType<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,
      IndirectCallableRelation<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<InputIterable Rng1, InputIterable Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
        Projected<IteratorType<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 Proj1 = identity, class Proj2 = identity,
      IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    tuple<I1, I2, O>
      set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
                Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
      class Proj1 = identity, class Proj2 = identity,
      IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
        Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
    requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
    tuple<IteratorType<Rng1>, IteratorType<Rng2>, 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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  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<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  pair<I1, O>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  pair<IteratorType<Rng1>, 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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tuple<I1, I2, O>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, 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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<Rng>
    sort_heap(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  bool
    is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<RandomAccessIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
```

```
  IteratorType<Rng>
    is_heap_until(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

// 25.4.7, minimum and maximum:
template<TotallyOrdered T>
  constexpr const T& min(const T& a, const T& b);

template<class T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr const T& min(const T& a, const T& b, Comp comp);
```

[Editor's note: REVIEW: The Palo Alto report returns by const reference here but the current standard returns by value.]

```
template<TotallyOrdered T>
  requires Semiregular<T>
  constexpr T min(initializer_list<T> t);

template<InputIterable Rng>
  requires TotallyOrdered<ValueType<IteratorType<Rng>>>() &&
    Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    min(Rng&& rng);

template<Semiregular T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr T min(initializer_list<T> t, Comp comp);

template<InputIterable Rng, IndirectCallableRelation<IteratorType<Rng>> Comp>
  requires Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    min(Rng&& rng, Comp comp);

template<TotallyOrdered T>
  constexpr const T& max(const T& a, const T& b);

template<class T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr const T& max(const T& a, const T& b, Comp comp);

template<TotallyOrdered T>
  requires Semiregular<T>
  constexpr T max(initializer_list<T> t);

template<InputIterable Rng>
  requires TotallyOrdered<ValueType<IteratorType<Rng>>>() &&
    Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    max(Rng&& rng);

template<Semiregular T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr T max(initializer_list<T> t, Comp comp);

template<InputIterable Rng, IndirectCallableRelation<IteratorType<Rng>> Comp>
  requires Semiregular<ValueType<IteratorType<Rng>>>
```

```
    ValueType<IteratorType<Rng>>
      max(Rng&& rng, Comp comp);

  template<TotallyOrdered T>
    constexpr pair<const T&, const T&>
      minmax(const T& a, const T& b);

  template<class T, class Comp>
    requires Relation<FunctionType<Comp>, T>
    constexpr pair<const T&, const T&>
      minmax(const T& a, const T& b, Comp comp);

  template<TotallyOrdered T>
    requires Semiregular<T>
    constexpr pair<T, T> minmax(initializer_list<T> t);

  template<InputIterable Rng>
    requires TotallyOrdered<ValueType<IteratorType<Rng>>>() &&
      Semiregular<ValueType<IteratorType<Rng>>>
    pair<ValueType<IteratorType<Rng>>, ValueType<IteratorType<Rng>>>
      minmax(Rng&& rng);

  template<Semiregular T, class Comp>
    requires Relation<FunctionType<Comp>, T>
    constexpr pair<T, T> minmax(initializer_list<T> t, Comp comp);

  template<InputIterable Rng, IndirectCallableRelation<IteratorType<Rng>> Comp>
    requires Semiregular<ValueType<IteratorType<Rng>>>
    pair<ValueType<IteratorType<Rng>>, ValueType<IteratorType<Rng>>>
      minmax(Rng&& rng, Comp comp);

  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
    I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template<ForwardIterable Rng, class Proj = identity,
      IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
    IteratorType<Rng>
      min_element(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
    I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template<ForwardIterable Rng, class Proj = identity,
      IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
    IteratorType<Rng>
      max_element(Rng& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
    pair<I, I>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template<ForwardIterable Rng, class Proj = identity,
```

```
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  pair<IteratorType<Rng>, IteratorType<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,
    IndirectCallableRelation<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<InputIterable Rng1, InputIterable Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<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 Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  requires Sortable<I, Comp, Proj>
  bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<BidirectionalIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  bool
    next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  requires Sortable<I, Comp, Proj>
  bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<BidirectionalIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  requires Sortable<IteratorType<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 (24.2.14). If an algorithm's template parameter

is `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the actual template argument shall satisfy the requirements of an output iterator (24.2.16). If an algorithm's template parameter is `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the actual template argument shall satisfy the requirements of a forward iterator (24.2.17). If an algorithm's template parameter is `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the actual template argument shall satisfy the requirements of a bidirectional iterator (24.2.18). If an algorithm's template parameter is `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the actual template argument shall satisfy the requirements of a random-access iterator (24.2.19).

6 If an algorithm's **Effects** section says that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall satisfy the requirements of a mutable iterator (24.2). [*Note:* This requirement does not affect arguments that are declared as `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable. — *end note*]

7 Both in-place and copying versions are provided for certain algorithms.[2] When such a version is provided for *algorithm* it is called *algorithm_copy*. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

8 The `Predicate` parameter is used whenever an algorithm expects a function object (20.9) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (Clause 4). The function object `pred` shall not apply any non-constant function through the dereferenced iterator.

9 The `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (Clause 4). `BinaryPredicate` always takes the first iterator's `value_type` as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (Clause 4). `binary_pred` shall not apply any non-constant function through the dereferenced iterators.

10 [*Note:* Projections and predicates are typically used as follows:

```
auto&& proj_ = callable(proj);
auto&& pred_ = callable(pred);
if(pred_(proj_(*first))) // ...
```

— *end note*]

11 [*Note:* Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.9.3), or some equivalent solution. — *end note*]

12 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either true or false in boolean contexts.

13 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

2) 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`.

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

and that of `b-a` is the same as of

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

14 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;
```

15 Overloads of algorithms that take `Iterable` arguments (24.10.2.2) behave as if they are implemented by calling `begin` and `end` on the `Iterable` and dispatching to the overload that takes separate iterator and sentinel arguments.

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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<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>
  pair<I, Fun>
    for_each(I first, S last, Fun f, Proj proj = Proj{});

template<InputIterable Rng, class Proj = identity,
    IndirectCallable<Projected<IteratorType<Rng>, Proj>> Fun>
  pair<IteratorType<Rng>, Fun>
    for_each(Rng& rng, Fun f, Proj proj = Proj{});
```

1 *Requires:* `Function` shall meet the requirements of `MoveConstructible` (~~Table 20~~ 19.2.11). [*Note:* `Function` need not meet the requirements of `CopyConstructible` (~~Table 21~~ 19.2.12). — *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 IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  I find(I first, S last, const T& value, Proj proj = Proj{});

template<InputIterable Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  IteratorType<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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  IteratorType<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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  IteratorType<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<ForwardIterable Rng1, ForwardIterable Rng2,
    class Proj = identity,
```

```
    IndirectCallableRelation<IteratorType<Rng2>,
      Projected<IteratorType<Rng>, Proj>> Pred = equal_to<>>
  IteratorType<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<InputIterable Rng1, ForwardIterable Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Pred = equal_to<>>
  IteratorType<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<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Pred = equal_to<>>
  IteratorType<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 IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  DifferenceType<I>
    count(I first, S last, const T& value, Proj proj = Proj{});

template<InputIterable Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  DifferenceType<IteratorType<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  DifferenceType<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  DifferenceType<IteratorType<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);

template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
    class Proj1 = identity, class Proj2 = identity
    IndirectCallablePredicate<Projected<I1, Proj1>, Projected<I2, Proj2>> Pred = equal_to<>>
  pair<I1, I2>
    mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, WeakInputIterator I2,
    class Proj1 = identity, class Proj2 = identity
    IndirectCallablePredicate<Projected<IteratorType<Rng1>, Proj1>,
      Projected<I2, Proj2>> Pred = equal_to<>>
  pair<IteratorType<Rng1>, 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<>>
  pair<I1, I2>
    mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Pred = equal_to<>>
  pair<IteratorType<Rng1>, IteratorType<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);

template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool equal(I1 first1, S1 last1,
             I2 first2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, WeakInputIterator I2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<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<InputIterable Rng1, InputIterable Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, IteratorType<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~~ condition~~s~~ 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~~ I1 and S1 model `SizedIteratorRange`, and I2 and S2 model `SizedIteratorRange`, 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);

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<ForwardIterable Rng1, ForwardIterator I2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<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<ForwardIterable Rng1, ForwardIterable Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, IteratorType<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~~ `I1` and `S1` model `SizedIteratorRange`, and `I2` and `S2` model `SizedIteratorRange`, 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<ForwardIterable Rng1, ForwardIterable Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, IteratorType<Rng2>, Pred, Proj1, Proj2>
  IteratorType<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<I1, const T*, Pred, Proj>
  I
    search_n(I first, S last, DifferenceType<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template<ForwardIterable Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<IteratorType<Rng1>, const T*, Pred, Proj>
  IteratorType<Rng>
    search_n(Rng& rng, DifferenceType<IteratorType<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>
  pair<I, O>
    copy(I first, S last, O result);
```

```
template<InputIterable Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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<WeakInputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  pair<I, O>
    copy_n(I first, iterator_distance_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>
  pair<I, O>
    copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});
```

```
template<InputIterable Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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.

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>
  pair<I1, I2>
    copy_backward(I1 first, I1 last, I2 result);

template<BidirectionalIterable Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<IteratorType<Rng>, I>
  pair<IteratorType<Rng>, 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`.[3] 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>
  pair<I, O>
    move(I first, S last, O result);

template<InputIterable Rng, WeaklyIncrementable O>
  requires IndirectlyMovable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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>
  pair<I1, I2>
    move_backward(I1 first, I1 last, I2 result);

template<BidirectionalIterable Rng, BidirectionalIterator I>
```

3) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

```
  requires IndirectlyMovable<IteratorType<Rng>, I>
  pair<IteratorType<Rng>, I>
    move_backward(Rng& rng, I result);
```

5 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result)` starting from `last - 1` and proceeding to first.[4] 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);

template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
  requires IndirectlySwappable<I1, I2>
  pair<I1, I2>
    swap_ranges(I1 first1, S1 last1, I2 first2);

template<ForwardIterable Rng, ForwardIterator I>
  requires IndirectlySwappable<IteratorType<Rng>, I>
  pair<IteratorType<Rng>, I>
    swap_ranges(Rng& rng1, I first2);

template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
  requires IndirectlySwappable<I1, I2>
  pair<I1, I2>
    swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template<ForwardIterable Rng1, ForwardIterable Rng2>
  requires IndirectlySwappable<IteratorType<Rng1>, IteratorType<Rng2>>
  pair<IteratorType<Rng1>, IteratorType<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 (19.2.17) `*(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 (19.2.17) `*b`.

4) `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, class Proj = identity,
    IndirectCallable<Projected<I, Proj>> F,
    WeakOutputIterator<IndirectCallableResultType<F, Projected<I, Proj>>> O>
  pair<I, O>
  transform(I first, S last, O result, F op, Proj proj = Proj{});

template<InputIterable Rng, class Proj = identity,
    IndirectCallable<Projected<IteratorType<Rng>, Proj>> F,
    WeakOutputIterator<IndirectCallableResultType<F,
      Projected<IteratorType<Rng>, Proj>>> O>
  pair<IteratorType<Rng>, O>
    transform(Rng& rng, O result, F op, Proj proj = Proj{});

template<InputIterator I1, Sentinel<I1> S1, WeakInputIterator I2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallable<Projected<I1, Proj1>, Projected<I2, Proj2>> F,
    WeakOutputIterator<IndirectCallableResultType<F, Projected<I1, Proj1>,
      Projected<I2, Proj2>>> O>
tuple<I1, I2, O>
    transform(I1 first1, S1 last1, I2 first2, O result,
              F binary_op, Proj1 proj1 = Proj1{}, proj2 proj2 = Proj2{});

template<InputIterable Rng, WeakInputIterator I,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallable<Projected<IteratorType<Rng>, Proj1>, Projected<I, Proj2>> F,
    WeakOutputIterator<IndirectCallableResultType<F,
      Projected<IteratorType<Rng>, Proj1>, Projected<I, Proj2>>> O>
  tuple<IteratorType<Rng>, I, 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,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallable<Projected<I1, Proj1>, Projected<I2, Proj2>> F,
    WeakOutputIterator<IndirectCallableResultType<F, Projected<I1, Proj1>,
      Projected<I2, Proj2>>> O>
  tuple<I1, I2, O>
  transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            F binary_op, Proj1 proj1 = Proj1{}, proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2,
```

```
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallable<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> F,
    WeakOutputIterator<IndirectCallableResultType<F,
      Projected<IteratorType<Rng1>, Proj1>, Projected<IteratorType<Rng2>, Proj2>>> O>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, 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*].[5]

4 *Returns:* ~~result + (last1 - first1)~~make_pair(first1 + *N*, result + *N*) or make_tuple(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, Semiregular T2, class Proj = identity>
  requires Writable<I, T2> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T1 *>
  I
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template<ForwardIterable Rng, class T1, Semiregular T2, class Proj = identity>
  requires Writable<IteratorType<Rng>, T2> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T1 *>
  IteratorType<Rng>
    replace(Rng& rng, const T1& old_value, const T2& new_value);

template<ForwardIterator I, Sentinel<I> S, Semiregular T, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires Writable<I, T>
  I
```

5) The use of fully closed ranges is intentional.

```
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template<ForwardIterable Rng, Semiregular T, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Writable<IteratorType<Rng>, T>
  IteratorType<Rng>
    replace_if(Rng& rng, Pred pred, const T& new_value, Proj proj = Proj{});
```

1 *Requires:* The expression `*first = new_value` shall be valid.

2 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first,last)` with `new_value`, when the following corresponding conditions hold: ~~`*i == old_value`~~`INVOKE(proj, *i) == old_value`, ~~`pred(*i) != false`~~`INVOKE(pred, INVOKE(proj, *i)) != false`.

3 *Returns:* `last`.

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

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator
    replace_copy(InputIterator first, InputIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
  OutputIterator
    replace_copy_if(InputIterator first, InputIterator last,
                    OutputIterator result,
                    Predicate pred, const T& new_value);

template<InputIterator I, Sentinel<I> S, class T1, Semiregular T2, WeakOutputIterator<T2> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T1 *>
  pair<I, O>
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template<InputIterable Rng, class T1, Semiregular T2, WeakOutputIterator<T2> O,
    class Proj = identity>
  requires IndirectlyCopyable<IteratorType<Rng>, O> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T1 *>
  pair<IteratorType<Rng>, O>
    replace_copy(Rng& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template<InputIterator I, Sentinel<I> S, Semiregular T, WeakOutputIterator<T> O,
    class Proj = identity, IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  pair<I, O>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template<InputIterable Rng, Semiregular T, WeakOutputIterator<T> O, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    replace_copy_if(Rng& rng, O result, Pred pred, const T& new_value,
```

```
                    Proj proj = Proj{});
```

5 *Requires:* ~~The results of the expressions `*first` and `new_value` shall be writable to the `result` output iterator.~~ The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

6 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false

INVOKE(proj, *(first + (i - result))) == old_value
INVOKE(pred, INVOKE(proj, *(first + (i - result)))) != false
```

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

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

25.3.6 Fill [alg.fill]

```
template<class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator last, const T& value);

template<class OutputIterator, class Size, class T>
  OutputIterator fill_n(OutputIterator first, Size n, const T& value);

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

template<Semiregular T, OutputIterable<T> Rng>
  IteratorType<Rng>
    fill(Rng& rng, const T& value);

template<Semiregular T, WeakOutputIterator<T> O>
  O fill_n(O first, DifferenceType<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<Function F, OutputIterator<ResultType<F>> O,
    Sentinel<O> S>
```

```
  O generate(O first, S last, F gen);

template<Function F, OutputIterable<ResultType<F>> Rng>
  IteratorType<Rng>
    generate(Rng& rng, F gen);

template<Function F, WeakOutputIterator<ResultType<F>> O>
  O generate_n(O first, DistanceType<O> n, F gen);
```

1 *Effects:* ~~The first algorithm~~generate invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,last)`. ~~The second algorithm~~generate_n invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing.

2 *Requires:* `gen` takes no arguments, `Size` shall be convertible to an integral type (4.7, 12.3).

3 *Returns:* `generate` returns `last`. `generate_n` returns `first + n` for non-negative values of `n` and `first` for negative values.

4 *Complexity:* Exactly `last - first`, `n`, or 0 invocations of `gen` and assignments, respectively.

25.3.8 Remove [alg.remove]

```
template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  I remove(I first, S last, const T& value, Proj proj = Proj{});

template<ForwardIterable Rng, class T, class Proj = identity>
  requires Permutable<IteratorType<Rng>> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  IteratorType<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<ForwardIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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> &&
    IndirectRelation<equal_to<>, Projected<I, Proj>, const T *>
  pair<I, O> remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<IteratorType<Rng>, O> &&
    IndirectRelation<equal_to<>, Projected<IteratorType<Rng>, Proj>, const T *>
  pair<IteratorType<Rng>, 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>
  pair<I, O>
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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`, *`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<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> R = equal_to<>>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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>
  pair<I, O>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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<BidirectionalIterable Rng>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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 (19.2.17).

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>
  pair<I, O> reverse_copy(I first, S last, O result);

template<BidirectionalIterable Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, 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>
  pair<I, I> rotate(I first, I middle, S last);
```

```
template<ForwardIterable Rng>
  requires Permutable<IteratorType<Rng>>
  pair<IteratorType<Rng>, IteratorType<Rng>>
    rotate(Rng& rng, IteratorType<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` (19.2.17). 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>
  pair<I, O>
    rotate_copy(I first, I middle, S last, O result);
```

```
template<ForwardIterable Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<IteratorType<Rng>, O>
  pair<IteratorType<Rng>, O>
    rotate_copy(Rng& rng, IteratorType<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, UniformRandomNumberGenerator Gen>
  requires Permutable<I> && Convertible<ResultType<Gen>, DifferenceType<I>>
  I shuffle(I first, S last, Gen&& g);
```

```
template<RandomAccessIterable Rng, UniformRandomNumberGenerator Gen>
  requires Permutable<I> && Convertible<ResultType<Gen>, DifferenceType<I>>
  IteratorType<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` (19.2.17). 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<InputIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<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<ForwardIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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 and 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` (19.2.17).

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<BidirectionalIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires Permutable<IteratorType<Rng>>
  IteratorType<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 and 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` (19.2.17). 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>
  tuple<I, O1, O2>
    partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<IteratorType<Rng>, O1> &&
```

```
    IndirectlyCopyable<IteratorType<Rng>, O2>
  tuple<IteratorType<Rng>, O1, 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` `OutputIterator`s, 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<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  requires IndirectlyMovable<I, O1> && IndirectlyMovable<I, O2>
  tuple<I, O1, O2>
    partition_move(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template<InputIterable Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  requires IndirectlyMovable<IteratorType<Rng>, O1> &&
    IndirectlyMovable<IteratorType<Rng>, O2>
  tuple<IteratorType<Rng>, O1, O2>
    partition_move(Rng& rng, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});
```

16 *Requires:* The input range shall not overlap with either of the output ranges.

17 *Effects:* For each iterator `i` in `[first,last)`, moves `*i` to the output range beginning with `out_true` if `INVOKE(pred, INVOKE(proj, *i))` is `true`, or to the output range beginning with `out_false` otherwise.

18 *Returns:* A tuple `p` such that `get<0>(p)` is `last`, `get<1>(p)` is the end of the output range beginning at `out_true` and `get<2>(p)` is the end of the output range beginning at `out_false`.

19 *Complexity:* Exactly `last - first` applications of `pred` and `proj`.

```
template<class ForwardIterator, class Predicate>
  ForwardIterator partition_point(ForwardIterator first,
                                  ForwardIterator last,
                                  Predicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectCallablePredicate<Projected<I, Proj>> Pred>
  I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallablePredicate<Projected<IteratorType<Rng>, Proj>> Pred>
  IteratorType<Rng>
    partition_point(Rng& rng, Pred pred, Proj proj = Proj{});
```

20 *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.

21 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

22 *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)~~an 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.

4 The term *strict* refers to the requirement of an irreflexive relation (~~`!comp(x, x)`~~`!`*`INVOKE`*`(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)`~~`!`*`INVOKE`*`(comp, a, b) && !`*`INVOKE`*`(comp, b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

(4.1) — ~~`comp(a, b) && comp(b, c)`~~*`INVOKE`*`(comp, a, b) && `*`INVOKE`*`(comp, b, c)` implies ~~`comp(a, c)`~~*`INVOKE`*`(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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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` (19.2.17). 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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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` (19.2.17). 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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<Rng>
    partial_sort(Rng& rng, IteratorType<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` (19.2.17). 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<I> S1, RandomAccessIterator I2, Sentinel<I> S2,
    class R = less<>, class Proj = identity>
  requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj>
  I2
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                      Comp comp = Comp{}, Proj proj = Proj{});

template<InputIterable Rng1, RandomAccessIterable Rng2, class R = less<>,
    class Proj = identity>
  requires IndirectlyCopyable<IteratorType<Rng1>, IteratorType<Rng2>> &&
      Sortable<IteratorType<Rng2>, Comp, Proj>
  IteratorType<Rng2>
```

```
partial_sort_copy(Rng1& rng, Rng2& result_rng, Comp comp = Comp{},
                  Proj proj = Proj{});
```

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` (19.2.17). 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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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, Proj proj = Proj{});

template<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<Rng>
    nth_element(Rng& rng, IteratorType<Rng> nth, 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` (19.2.17). 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, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  I
    lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
```

```
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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 conditions hold: ~~`*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, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  I
    upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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 conditions hold: ~~`!(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, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  pair<I, I>
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  pair<IteratorType<Rng>, IteratorType<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, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<I, Proj>> Comp = less<>>
  bool
    binary_search(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

template<ForwardIterable Rng, TotallyOrdered T, class Proj = identity,
    IndirectCallableRelation<const T *, Projected<IteratorType<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 $\log_2$(`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,
    Incrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tuple<I1, I2, O>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, Incrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, 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~~ [Editor's note: TODO The following postcondition isn't well-formed:] `is_sorted(result, result_last, comp)`~~, respectively~~.

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_tuple(last1, last2, result + (last1 - first1) + (last2 - first2))`.

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<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    Incrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires MergeMovable<I1, I2, O, Comp, Proj1, Proj2>
  tuple<I1, I2, O>
```

```
    merge_move(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, Incrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires MergeMovable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, O>
    merge_move(Rng1& rng1, Rng2& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

6 *Effects:* Moves 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 [Editor's note: TODO The following postcondition isn't well-formed:] `is_sorted(result, result_last, comp)`.

7 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `comp`, `proj1`, and `proj2`. The resulting range shall not overlap with either of the original ranges.

8 *Returns:* `make_tuple(last1, last2, result + (last1 - first1) + (last2 - first2))`.

9 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` applications of the comparison function and each projection.

10 *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<BidirectionalIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<Rng>
    inplace_merge(Rng& rng, IteratorType<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});
```

11 *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.

12 *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` (19.2.17). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22)~~.

13 *Returns:* `last`

14 *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.

15 *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 `multisets` (23.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multisets` in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

25.4.5.1 `includes` [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,
    IndirectCallableRelation<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<InputIterable Rng1, InputIterable Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tuple<I1, I2, O>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, 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_tuple(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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  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<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  pair<I1, O>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  pair<IteratorType<Rng1>, 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~~`make_pair(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 Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<I1, Proj1>, Projected<I2, Proj2>> Comp = less<>>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tuple<I1, I2, O>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template<InputIterable Rng1, InputIterable Rng2, WeaklyIncrementable O,
    class Proj1 = identity, class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<Rng2>, Proj2>> Comp = less<>>
  requires Mergeable<IteratorType<Rng1>, IteratorType<Rng2>, O, Comp, Proj1, Proj2>
  tuple<IteratorType<Rng1>, IteratorType<Rng2>, 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_tuple(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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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` (19.2.17). 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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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<RandomAccessIterable Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<IteratorType<Rng>, Comp, Proj>
  IteratorType<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` (19.2.17). 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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template<RandomAccessIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template<RandomAccessIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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<TotallyOrdered T>
  constexpr const T& min(const T& a, const T& b);

template<class T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr const T& min(const T& a, const T& b, Comp comp);
```

1 *Requires:* Type `T` is `LessThanComparable` (Table 18).

2 *Returns:* The smaller value.

3 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
  constexpr T min(initializer_list<T> t);
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp);

template<TotallyOrdered T>
  requires Semiregular<T>
  constexpr T min(initializer_list<T> rng);

template<InputIterable Rng>
  requires TotallyOrdered<ValueType<IteratorType<Rng>>>() &&
    Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    min(Rng&& rng);

template<Semiregular T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr T min(initializer_list<T> rng, Comp comp);

template<InputIterable Rng, IndirectCallableRelation<IteratorType<Rng>> Comp>
  requires Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    min(Rng&& rng, Comp comp);
```

4 *Requires:* ~~`T` is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`~~`distance(begin(rng), end(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<TotallyOrdered T>
  constexpr const T& max(const T& a, const T& b);

template<class T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr const T& max(const T& a, const T& b, Comp comp);
```

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<TotallyOrdered T>
  requires Semiregular<T>
  constexpr T max(initializer_list<T> rng);

template<InputIterable Rng>
  requires TotallyOrdered<ValueType<IteratorType<Rng>>>() &&
    Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    max(Rng&& rng);

template<Semiregular T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr T max(initializer_list<T> rng, Comp comp);

template<InputIterable Rng, IndirectCallableRelation<IteratorType<Rng>> Comp>
  requires Semiregular<ValueType<IteratorType<Rng>>>
  ValueType<IteratorType<Rng>>
    max(Rng&& rng, Comp comp);
```

10 *Requires:* ~~`T` is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`~~`distance(begin(rng), end(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<TotallyOrdered T>
  constexpr pair<const T&, const T&>
    minmax(const T& a, const T& b);

template<class T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr pair<const T&, const T&>
    minmax(const T& a, const T& b, Comp comp);
```

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.

```
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<TotallyOrdered T>
  requires Semiregular<T>
  constexpr pair<T, T> minmax(initializer_list<T> rng);

template<InputIterable Rng>
  requires TotallyOrdered<ValueType<IteratorType<Rng>>>() &&
    Semiregular<ValueType<IteratorType<Rng>>>
  pair<ValueType<IteratorType<Rng>>, ValueType<IteratorType<Rng>>>
    minmax(Rng&& rng);

template<Semiregular T, class Comp>
  requires Relation<FunctionType<Comp>, T>
  constexpr pair<T, T> minmax(initializer_list<T> rng, Comp comp);

template<InputIterable Rng, IndirectCallableRelation<IteratorType<Rng>> Comp>
  requires Semiregular<ValueType<IteratorType<Rng>>>
  pair<ValueType<IteratorType<Rng>>, ValueType<IteratorType<Rng>>>
    minmax(Rng&& rng, Comp comp);
```

17 *Requires:* ~~`T` is `LessThanComparable` and `CopyConstructible` and `t.size() > 0`~~`distance(begin(rng), end(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(begin(rng), end(rng))` applications of the corresponding predicate.

```
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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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 conditions hold: ~~`!(*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 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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  IteratorType<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 conditions hold: ~~`!(*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 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,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  pair<I, I>
    minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<ForwardIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  pair<IteratorType<Rng>, IteratorType<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 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,
    IndirectCallableRelation<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<InputIterable Rng1, InputIterable Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng1>, Proj1>,
      Projected<IteratorType<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 Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  requires Sortable<I, Comp, Proj>
  bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<BidirectionalIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  requires Sortable<IteratorType<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` (19.2.17).

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 Proj = identity,
    IndirectCallableRelation<Projected<I, Proj>> Comp = less<>>
  requires Sortable<I, Comp, Proj>
  bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template<BidirectionalIterable Rng, class Proj = identity,
    IndirectCallableRelation<Projected<IteratorType<Rng>, Proj>> Comp = less<>>
  requires Sortable<IteratorType<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` (19.2.17).

7 *Complexity:* At most `(last - first)/2` swaps.

25.5 C library algorithms [alg.c.library]

1 Table 7 describes some of the contents of the header `<cstdlib>`.

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 *));
```

Table 7 — Header <cstdlib> synopsis

Type	Name(s)	
Type:	size_t	
Functions:	bsearch	qsort

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.

Annex A (informative) Acknowledgements [acknowledgements]

1 I would like to thank Andrew Sutton for his work on the Concepts TS (1.2), for his help formalizing the ideas of the range-v3 library [2] on which this work is based, and for his review of this document.

2 Sean Parent has made major contributions to both the foundations and the wording of this peper.

3 Stephan T. Lavavej offered a careful review of much of this document, a non-trivial undertaking.

4 I would also like to thank the members of the Ranges SG who offered feedback on early drafts; especially, Tony Van Eerd, Casey Carter, and Walter Brown.

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 `TotallyOrdered` 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 `TotallyOrdered`). Similar constraints are added to the other function objects in `<functional>`. Also, the so-called transparent relational function objects (`less<void>` and friends) have their return type coerced to `bool`. As with the coarsely-grained algorithm constraints, these function object constraints are likely to cause user code to break.

2 Real-world experience is needed to assess the seriousness of the breakage. From a correctness point of view, the constraints are logical and valuable, but it's possible that for the sake of compatibility we provide both constrained and unconstrained functional objects.

B.1.4 Iterators and Default-Constructibility [diff.cpp.defaultconstruct]

1 In today's STL, iterators need not be default-constructible. The `Iterator` concept described in this document requires default-constructibility. This could potentially cause breakage in users' code. Also, it makes the implementation of some types of iterators more complicated. Any iterator that has members that are not default constructible (e.g., an iterator that contains a lambda that has captured by reference) must take special steps to provide default-constructibility (e.g., by wrapping non-default-constructible types in `std::optional`). This can weaken class invariants.

2 The guarantee of default-constructibility simplifies the implementation of much iterator- and range-based code that would otherwise need to wrap iterators in `std::optional`. But the needs of backward-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 `WeakInputIterator`.

2 This breakage is intentional and inherent in the decomposition of the iterator concepts established by The Palo Alto report (1.2).

B.2 Ranges and the Palo Alto TR (N3351) [diff.n3351]

1 The Palo Alto report (1.2) presents a comprehensive design for the Standard Template Library constrained with concepts. It served both as a basis for the Concepts Lite language feature and for this document. However, this document diverges from the Palo Alto report in small ways. The differences are in the interests of backwards compatability, to avoid confusing a large installed base of programmers already familiar with the STL, and to keep the scope of this document as small as possible. This section describes the ways in which the two suggested designs differ.

B.2.1 Sentinels [diff.n3351.sentinels]

1 In the design presented in this document, the type of a range's end delimitter may differ from the iterator representing the range's start position. The reasons for this change are discribed in N4128 (1.2). This causes a number of difference from the Palo Alto report:

(1.1) — The algorithms get an additional constraint for the sentinel.

(1.2) — The return types of the algorithms are changed as described above (B.1.1).

(1.3) — Some algorithms have operational semantics that require them to know the physical end position (e.g., `reverse`). Those algorithms must make an $\mathcal{O}(N)$ 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 `SizedIteratorRange` (24.5.1). The `SizedIteratorRange` concept requires that `b - a` return the number of times `a` would have to be incremented to make it compare equal to `b`. Any type purporting to be a random access iterator that fails to meet that requirement is by definition not a valid random access iterator.

B.2.5 Output Iterators [diff.n3351.output_iters]

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 and the more freequently-needed `WeakOutputIterator`. Even if they are not strictly needed, their absence would be surprising.

B.2.6 No Algorithm Reformulations [diff.n3351.no_eop_algos]

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 [4]). The algorithms presented in The Palo Alto report reflect the results of that research in the algorithm constraints, some of which (e.g., `sort`, `inplace_merge`) take iterators with weaker categories than they do in the current standard. The design presented in this document does not reflect those changes. Although those changes are desirable, generalizing the algoritms 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 Implementation Experience with Concepts [future.experience]

1 The ideas presented here have been vetted in the range-v3 ([2]) library, which has seen both heavy development and heavy use over the past year and a half. However, this library is implemented in C++11 with the help of a library-based emulation layer for Concepts Lite. Andrew Sutton's origin ([6]) library implements many of these ideas using real concepts, but it's a subset of the work presented here. A critically important piece of this work will be to fully implement this design in C++17 with a compiler that supports Concepts.

C.2 Algorithms and Rvalue Ranges [future.rvalrngs]

1 As presented in this document, the vast majority of algorithms do not accept rvalue Iterables. This shortcoming becomes more painful with the addition of range views (C.5). The issue is one of safety. Those algorithms that return iterators into the input range, like `find`, can return iterators that will immediately be made invalid once the temporary Iterable they point into gets destroyed – but not for *all* Iterables. For an Iterable that is little more than a pair of iterators, it is perfectly safe to use an iterator into the Iterable after the Iterable has been destroyed. The iterator's validity is not tied to the Iterable's lifetime at all.

2 The range-v3 library ([2]) on which this proposal is based has recently implemented an experimental solution to this problem. The library provides a wrapper called `dangling` that the algorithms can use to mark returned iterators that point into rvalue ranges. It works as follows:

```
template <class T>
struct dangling {
  dangling() = default;
  dangling(T t) : t_(t) {}             // implicit
  T get_unsafe() const { return t_; } // explicit
private:
  T t_;
};

template <Iterable Rng>
using safe_iterator_t =
  std::conditional_t<
    std::is_lvalue_reference<Rng>::value,
    IteratorType<Rng>,
    dangling<IteratorType<Rng>>>;

template <InputIterable Rng, class T, class Proj = identity>
  requires //...
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{}) {
      return find(begin(rng), end(rng), value, std::move(proj));
    }
```

3 Of particular note is that a `dangling<Iterator>` is *not* an iterator. In order to obtain an iterator into an expired Iterable, the user must explicitly call a method called `get_unsafe`. That makes it impossible for a user to accidentally use an iterator that may be invalid.

4 An extension of this idea would be to define `safe_iterator_t` in terms of a trait that can be specialized on user-defined Iterable types. That way, Iterables that guarantee their iterators can outlive the Iterable will never have their iterators wrapped in `dangling`.

5 If time shows that the range-v3 solution is workable, a future paper can integrate the new functionality into this document.

C.3 Proxy Iterators [future.proxy]

1 As early at 1998 when Herb Sutter published his "When is a Container not a Container" [5] article, it was known that proxy iterators were a challenge that the current iterator concept hierarchy could not meet. The problem stems from the fact that the ForwardIterator concept as specified in the current standard requires the iterator's reference type to be a true reference, not a proxy. The Palo Alto report lifts this restriction in its respecification of the iterator concepts but doesn't actually solve the problem. The majority of algorithms, once you study the constraints, do not accept many interesting proxy iterator types.

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 [3].) This document does not reflect the results of that research.

Whether and how to best support proxy iterators is left as future work.

C.4 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.5 Range Views and Actions [future.views]

1 The vast majority of the power of ranges comes from their composability. *Views* on existing Iterables 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.6 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.7 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 `DifferenceType`. The `DifferenceType` must be a built-in integral type, and it must be large enough to hold the distance between any two valid iterators. This is implicit in the fact that the `distance` iterator primitive returns objects of type `DifferenceType`.

3 Given the fact that there are no infinite precision built-in integral types, the presence of `DifferenceType` in the concept requirements places a hard upper limit on how big ranges can be. This is a severe limitation.

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 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 `DifferenceType`, 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.8 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.9 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.10 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.

Bibliography

[1] Adobe source libraries. http://stlab.adobe.com. Accessed: 2014-10-8.

[2] Eric Niebler. Range-v3. https://github.com/ericniebler/range-v3. Accessed: 2015-4-6.

[3] 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.

[4] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[5] Herb Sutter. When is a container not a container? *C++ Report*, 14, 5 1999. http://www.gotw.ca/publications/mill09.htm.

[6] Andrew Sutton. Origin. https://github.com/asutton/origin. Accessed: 2015-4-6.

[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