Document Number: N4685
Date: 2017-07-31
Reply to: Eric Niebler
eric.niebler@gmail.com
Reply to: Casey Carter
casey@carter.net

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

Contents ii

List of Tables iv

1 Scope 1

2 References 2

3 Terms and definitions 3

4 General principles 4
4.1 Implementation compliance . . . 4
4.2 Namespaces, headers, and modifications to standard classes . . . 4

5 Statements 5
5.1 Iteration statements . . . 5

6 Library introduction 7
6.1 General . . . 7
6.2 Method of description (Informative) . . . 7
6.3 Library-wide requirements . . . 9

7 Concepts library 12
7.1 General . . . 12
7.2 Header `<experimental/ranges/concepts>` synopsis . . . 13
7.3 Core language concepts . . . 16
7.4 Comparison concepts . . . 21
7.5 Object concepts . . . 23
7.6 Callable concepts . . . 24

8 General utilities library 27
8.1 General . . . 27
8.2 Utility components . . . 27
8.3 Function objects . . . 29
8.4 Metaprogramming and type traits . . . 32
8.5 Tagged tuple-like types . . . 37

9 Iterators library 43
9.1 General . . . 43
9.2 Header `<experimental/ranges/iterator>` synopsis . . . 43
9.3 Iterator requirements . . . 52
9.4 Indirect callable requirements . . . 62
9.5 Common algorithm requirements . . . 64
9.6 Iterator primitives . . . 66
9.7 Iterator adaptors . . . 71
9.8 Stream iterators . . . 101

10 Ranges library 110
10.1 General 110
10.2 decay_copy 110
10.3 Header `<experimental/ranges/range>` synopsis 110
10.4 Range access 111
10.5 Range primitives 114
10.6 Range requirements 115

11 Algorithms library 119
11.1 General 119
11.2 Tag specifiers 138
11.3 Non-modifying sequence operations 138
11.4 Mutating sequence operations 144
11.5 Sorting and related operations 155

12 Numerics library 170
12.1 Uniform random number generator requirements 170

A Compatibility features 171
A.1 Rvalue range access 171
A.2 Range-and-a-half algorithms 171

B Acknowledgements 174

C Compatibility 175
C.1 C++ and Ranges 175
C.2 Ranges and the Palo Alto TR (N3351) 176

Bibliography 178

Index 179

Index of library names 180

List of Tables

1 Library categories 7
2 Ranges TS library headers 9

3 Fundamental concepts library summary 12

4 General utilities library summary 27
5 Additional type property predicates 34
6 Other transformations 35

7 Iterators library summary 43
8 Relations among iterator categories 52

9 Ranges library summary 110
10 Relations among range categories 115

11 Algorithms library summary 119

1 Scope [intro.scope]

> 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 This Technical Specification describes extensions to the C++ Programming Language (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 one core language facility. In particular, changes and extensions to the Standard Library include:

(1.1) — The formulation of the foundational and iterator concept requirements using the syntax of the Concepts TS (2).

(1.2) — Analogues of the Standard Library algorithms specified in terms of the new concepts.

(1.3) — The loosening of the algorithm constraints to permit the use of *sentinels* to denote the end of a range and corresponding changes to algorithm return types where necessary.

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

(1.5) — New algorithm overloads that take range objects.

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

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

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

(1.9) — Constrained analogues of the standard iterator adaptors and stream iterators that satisfy 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 (10.4).

3 This document does not specify constrained analogues of other parts of the Standard Library (e.g., the numeric algorithms), nor does it add range support to all the places that could benefit from it (e.g., the containers).

4 This document does not specify any new range views, actions, or facade or adaptor utilities; all are left as future work.

2 References [intro.refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

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

ISO/IEC 14882:2014 is herein called the *C++ Standard* and ISO/IEC TS 19217:2015 is called the *Concepts TS*.

3 Terms and definitions [intro.defs]

For the purposes of this document, the terms and definitions given in ISO/IEC 14882:2014 and the following apply.

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

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

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

3.1 [defns.const.subexpr]

constant subexpression
expression whose evaluation as subexpression of a conditional-expression `CE` (ISO/IEC 14882:2014 §5.16) would not prevent `CE` from being a core constant expression (ISO/IEC 14882:2014 §5.19)

3.2 [defns.expr.equiv]

expression-equivalent
relationship that exists between two expressions `E1` and `E2` such that

— `E1` and `E2` have the same effects,

— `noexcept(E1) == noexcept(E2)`, and

— `E1` is a constant subexpression if and only if `E2` is a constant subexpression

3.3 [defns.projection]

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

[*Example:*

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

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

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

— *end example*]

4 General principles [intro]

4.1 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in ISO/IEC 14882:2014 §1.4. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

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

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

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

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

2 The International Standard, ISO/IEC 14882, together with ISO/IEC TS 19217:2015 (the Concepts TS), provide important context and specification for this document. In places, this document suggests changes to be made to components in namespace `std` in-place. In other places, entire chapters and sections are copied from ISO/IEC 14882 and modified so as to define similar but different components in namespace `std::experimental::ranges::v1`.

3 Instructions to modify or add paragraphs are written as explicit instructions. Modifications made to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

4 This document assumes that the contents of the `std::experimental::ranges::v1` namespace will become a new constrained version of the C++ Standard Library that will be delivered alongside the existing unconstrained version.

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

6 New header names are prefixed with `experimental/ranges/`. Where the final element of a new header name is the same as an existing standard header name (e.g., `<experimental/ranges/algorithm>`), the new header shall include the standard header as if by

```
#include <algorithm>
```

5 Statements [stmt]

5.1 Iteration statements [stmt.iter]

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

[Editor's note: Modify ISO/IEC 14882:2014 §6.5.4/p1 to allow differently typed begin and end iterators, like in C++17.]

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
  }
}
```

The range-based `for` statement

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

is equivalent to

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

where

(1.1) — if the *for-range-initializer* is an *expression*, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two *init-declarators*);

(1.2) — `__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:~~

(1.3) — *begin-expr* and *end-expr* are determined as follows:

(1.3.1) — if ~~_RangeT~~the *for-range-initializer* is an expression of array type `R`, *begin-expr* and *end-expr* are `__range` and `__range + __bound`, respectively, where `__bound` is the array bound. If ~~_RangeT~~`R` is an array of unknown ~~size~~bound or an array of incomplete type, the program is ill-formed;

(1.3.2) — if ~~_RangeT~~the *for-range-initializer* is ~~a~~an expression of class type `C`, the *unqualified-ids* `begin` and `end` are looked up in the scope of ~~class _RangeT~~`C` 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;

(1.3.3) — 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*]

2 In the *decl-specifier-seq* of a *for-range-declaration*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`. The *decl-specifier-seq* shall not define a class or enumeration.

6 Library introduction [library]

6.1 General [library.general]

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

2 Clause 6.3, Clauses 7 through 12, and Annex A specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.

3 Detailed specifications for each of the components in the library are in Clauses 7–12, as shown in Table 1.

Table 1 — Library categories

Clause	Category
7	Concepts library
8	General utilities library
9	Iterators library
10	Ranges library
11	Algorithms library
12	Numerics library

4 The concepts library (Clause 7) describes library components that C++ programs may use to perform compile-time validation of template parameters and perform function dispatch based on properties of types.

5 The general utilities library (Clause 8) includes components used by other library elements and components used as infrastructure in C++ programs, such as function objects.

6 The iterators library (Clause 9) describes components that C++ programs may use to perform iterations over containers (Clause ISO/IEC 14882:2014 §23), streams (ISO/IEC 14882:2014 §27.7), stream buffers (ISO/IEC 14882:2014 §27.6), and ranges (10).

7 The ranges library (Clause 10) describes components for dealing with ranges of elements.

8 The algorithms library (Clause 11) describes components that C++ programs may use to perform algorithmic operations on containers (Clause ISO/IEC 14882:2014 §23) and other sequences.

9 The numerics library (Clause 12) provides concepts that are useful to constrain numeric algorithms.

6.2 Method of description (Informative) [description]

1 This subclause describes the conventions used to specify the Ranges library. 6.2.1 describes the structure of the normative Clauses 7 through 12 and Annex A. 6.2.2 describes other editorial conventions.

6.2.1 Structure of each clause [structure]

6.2.1.1 Elements [structure.elements]

1 Each library clause contains the following elements, as applicable:[1]

(1.1) — Summary

(1.2) — Requirements

(1.3) — Detailed specifications

1) To save space, items that do not apply to a Clause are omitted. For example, if a Clause does not specify any requirements, there will be no “Requirements” subclause.

6.2.1.2 Summary [structure.summary]

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

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

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

(3.1) — macros

(3.2) — values

(3.3) — types

(3.4) — classes and class templates

(3.5) — functions and function templates

(3.6) — objects

(3.7) — concepts

6.2.1.3 Requirements [structure.requirements]

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

(1.1) — Template arguments

(1.2) — Derived classes

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

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

3 Requirements are stated in terms of concepts (Concepts TS [dcl.spec.concept]). Concepts are stated in terms of well-defined expressions that define valid terms of the types that satisfy the concept. For every set of well-defined expression requirements there is a named concept that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 11) that uses the well-defined expression requirements is described in terms of the valid expressions for its formal type parameters.

4 Template argument requirements are sometimes referenced by name. See ISO/IEC 14882:2014 §17.5.2.1.

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

6 Required operations of any concept defined in this document need not be total functions; that is, some arguments to a required operation may result in the required semantics failing to be satisfied. [*Example:* The required `<` operator of the `StrictTotallyOrdered` concept (7.4.4) does not meet the semantic requirements of that concept when operating on NaNs. — *end example*] This does not affect whether a type satisfies the concept.

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

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

6.2.1.4 Detailed specifications [structure.specifications]

1 The detailed specifications of each entity defined in Clauses 7–12 follow the conventions established by ISO/IEC 14882:2014 §17.5.1.4.

6.2.2 Other conventions [conventions]

1 This subclause describes several editorial conventions used to describe the contents of the Ranges library. These conventions are for describing member functions (6.2.2.1), and private members (6.2.2.2).

6.2.2.1 Functions within classes [functions.within.classes]

1 This document follows the same conventions as specified in ISO/IEC 14882:2014 §17.5.2.2.

6.2.2.2 Private members [objects.within.classes]

1 This document follows the same conventions as specified in ISO/IEC 14882:2014 §17.5.2.3.

6.3 Library-wide requirements [requirements]

1 This subclause specifies requirements that apply to the entire Ranges library. Clauses 7 through 12 and Annex A specify the requirements of individual entities within the library.

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

3 Within this subclause, 6.3.1 describes the library's contents and organization, 6.3.3 describes how well-formed C++ programs gain access to library entities, 6.3.4 describes constraints on well-formed C++ programs, and 6.3.5 describes constraints on conforming implementations.

6.3.1 Library contents and organization [organization]

1 6.3.1.1 describes the entities and macros defined in the Ranges library.

6.3.1.1 Library contents [contents]

1 The Ranges library provides definitions for the entities and macros specified in the Ranges library headers (6.3.2).

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

6.3.2 Headers [headers]

1 Each element of the Ranges library is declared or defined (as appropriate) in a header.

2 The Ranges library provides the Ranges library headers, shown in Table 2.

Table 2 — Ranges TS library headers

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

6.3.3 Using the library [using]

6.3.3.1 Overview [using.overview]

1 This section describes how a C++ program gains access to the facilities of the Ranges library. 6.3.3.2 describes effects during translation phase 4, while 6.3.3.3 describes effects during phase 8 (ISO/IEC 14882:2014 §2.2).

6.3.3.2 Headers [using.headers]

The entities in the Ranges library are defined in headers, the use of which is governed by the same requirements as specified in ISO/IEC 14882:2014 §17.6.2.2.

6.3.3.3 Linkage [using.linkage]

1 Entities in the C++ standard library have external linkage (ISO/IEC 14882:2014 §3.5). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (ISO/IEC 14882:2014 §7.5).

6.3.4 Constraints on programs [constraints]

6.3.4.1 Overview [constraints.overview]

1 This section describes restrictions on C++ programs that use the facilities of the Ranges library. The following subclauses specify constraints on the program's use of Ranges library classes as base classes (6.3.4.2) and other constraints.

6.3.4.2 Derived classes [derived.classes]

1 Virtual member function signatures defined for a base class in the Ranges library may be overridden in a derived class defined in the program (ISO/IEC 14882:2014 §10.3).

6.3.4.3 Other functions [res.on.functions]

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

2 In particular, the effects are undefined if an incomplete type (ISO/IEC 14882:2014 §3.9) is used as a template argument when instantiating a template component or evaluating a concept, unless specifically allowed for that component.

6.3.4.4 Function arguments [res.on.arguments]

1 The constraints on arguments passed to C++ standard library function as specified in ISO/IEC 14882:2014 §17.6.4.9 also apply to arguments passed to functions in the Ranges library.

6.3.4.5 Library object access [res.on.objects]

1 The constraints on object access by C++ standard library functions as specified in ISO/IEC 14882:2014 §17.6.4.10 also apply to object access by functions in the Ranges library.

6.3.4.6 Requires paragraph [res.on.required]

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

6.3.4.7 Semantic requirements [res.on.requirements]

1 If the semantic requirements of a declaration's constraints (6.2.1.3) are not satisfied at the point of use, the program is ill-formed, no diagnostic required.

6.3.5 Conforming implementations [conforming]

1 The constraints upon, and latitude of, implementations of the Ranges library follow the same constraints and latitudes for implementations of the C++ standard library as specified in ISO/IEC 14882:2014 §17.6.5.

6.3.5.1 Customization Point Objects [customization.point.object]

1 A *customization point object* is a function object (8.3) with a literal class type that interacts with user-defined types while enforcing semantic requirements on that interaction.

2 The type of a customization point object shall satisfy `Semiregular` (7.5.3).

3 All instances of a specific customization point object type shall be equal (7.1.1).

4 The type of a customization point object `T` shall satisfy `Invocable<const T, Args...>` (7.6.2) when the types of `Args...` meet the requirements specified in that customization point object's definition. Otherwise, `T` shall not have a function call operator that participates in overload resolution.

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

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

7 [*Note:* Many of the customization point objects in the library evaluate function call expressions with an unqualified name which results in a call to a user-defined function found by argument dependent name lookup (ISO/IEC 14882:2014 §3.4.2). To preclude such an expression resulting in a call to unconstrained functions with the same name in namespace `std`, customization point objects specify that lookup for these expressions is performed in a context that includes deleted overloads matching the signatures of overloads defined in namespace `std`. When the deleted overloads are viable, user-defined overloads must be more specialized (ISO/IEC 14882:2014 §14.5.6.2) or more constrained (Concepts TS [temp.constr.order]) to be used by a customization point object. — *end note*]

7 Concepts library [concepts.lib]

7.1 General [concepts.lib.general]

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

2 The following subclauses describe core language concepts, comparison concepts, object concepts, and function concepts as summarized in Table 3.

Table 3 — Fundamental concepts library summary

	Subclause	Header(s)
7.3	Core language concepts	`<experimental/ranges/concepts>`
7.4	Comparison concepts	
7.5	Object concepts	
7.6	Callable concepts	

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

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

2 Not all input values must be valid for a given expression; e.g., for integers `a` and `b`, the expression `a / b` is not well-defined when `b` is `0`. This does not preclude the expression `a / b` being equality preserving. The *domain* of an expression is the set of input values for which the expression is required to be well-defined.

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

4 Expressions declared in a *requires-expression* in this document are required to be equality preserving, except for those annotated with the comment "not required to be equality preserving." An expression so annotated may be equality preserving, but is not required to be so.

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

6 Where a *requires-expression* declares an expression that is non-modifying for some constant lvalue operand, additional variations of that expression that accept a non-constant lvalue or (possibly constant) rvalue for the given operand are also required except where such an expression variation is explicitly required with

differing semantics. These *implicit expression variations* must meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of the variations is unspecified.

[*Example:*

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

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

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

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

had been declared as well.

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

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

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

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

7.2 Header `<experimental/ranges/concepts>` synopsis [concepts.lib.synopsis]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 7.3, core language concepts:
  // 7.3.2, Same:
  template <class T, class U>
  concept bool Same = see below;

  // 7.3.3, DerivedFrom:
  template <class T, class U>
  concept bool DerivedFrom = see below;

  // 7.3.4, ConvertibleTo:
  template <class T, class U>
  concept bool ConvertibleTo = see below;
```

```
// 7.3.5, CommonReference:
template <class T, class U>
concept bool CommonReference = see below;

// 7.3.6, Common:
template <class T, class U>
concept bool Common = see below;

// 7.3.7, Integral:
template <class T>
concept bool Integral = see below;

// 7.3.8, SignedIntegral:
template <class T>
concept bool SignedIntegral = see below;

// 7.3.9, UnsignedIntegral:
template <class T>
concept bool UnsignedIntegral = see below;

// 7.3.10, Assignable:
template <class T, class U>
concept bool Assignable = see below;

// 7.3.11, Swappable:
template <class T>
concept bool Swappable = see below;

template <class T, class U>
concept bool SwappableWith = see below;

// 7.3.12, Destructible:
template <class T>
concept bool Destructible = see below;

// 7.3.13, Constructible:
template <class T, class... Args>
concept bool Constructible = see below;

// 7.3.14, DefaultConstructible:
template <class T>
concept bool DefaultConstructible = see below;

// 7.3.15, MoveConstructible:
template <class T>
concept bool MoveConstructible = see below;

// 7.3.16, CopyConstructible:
template <class T>
concept bool CopyConstructible = see below;

// 7.4, comparison concepts:
// 7.4.2, Boolean:
template <class B>
concept bool Boolean = see below;
```

```
// 7.4.3, EqualityComparable:
template <class T, class U>
concept bool WeaklyEqualityComparableWith = see below;

template <class T>
concept bool EqualityComparable = see below;

template <class T, class U>
concept bool EqualityComparableWith = see below;

// 7.4.4, StrictTotallyOrdered:
template <class T>
concept bool StrictTotallyOrdered = see below;

template <class T, class U>
concept bool StrictTotallyOrderedWith = see below;

// 7.5, object concepts:
// 7.5.1, Movable:
template <class T>
concept bool Movable = see below;

// 7.5.2, Copyable:
template <class T>
concept bool Copyable = see below;

// 7.5.3, Semiregular:
template <class T>
concept bool Semiregular = see below;

// 7.5.4, Regular:
template <class T>
concept bool Regular = see below;

// 7.6, callable concepts:
// 7.6.2, Invocable:
template <class F, class... Args>
concept bool Invocable = see below;

// 7.6.3, RegularInvocable:
template <class F, class... Args>
concept bool RegularInvocable = see below;

// 7.6.4, Predicate:
template <class F, class... Args>
concept bool Predicate = see below;

// 7.6.5, Relation:
template <class R, class T, class U>
concept bool Relation = see below;

// 7.6.6, StrictWeakOrder:
template <class R, class T, class U>
concept bool StrictWeakOrder = see below;
```

```
}}}}
```

7.3 Core language concepts [concepts.lib.corelang]

7.3.1 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.

7.3.2 Concept `Same` [concepts.lib.corelang.same]

```
template <class T, class U>
concept bool Same = is_same<T, U>::value; // see below
```

1 There need not be any subsumption relationship between `Same<T, U>` and `is_same<T, U>::value`.

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

7.3.3 Concept `DerivedFrom` [concepts.lib.corelang.derived]

```
template <class T, class U>
concept bool DerivedFrom =
  is_base_of<U, T>::value &&
  is_convertible<remove_cv_t<T>*, remove_cv_t<U>*>::value; // see below
```

1 There need not be any subsumption relationship between `DerivedFrom<T, U>` and either `is_base_of<U, T>::value` or `is_convertible<remove_cv_t<T>*, remove_cv_t<U>*>::value`.

2 [*Note:* `DerivedFrom<T, U>` is satisfied if and only if `T` is publicly and unambiguously derived from `U`, or `T` and `U` are the same class type ignoring *cv*-qualifiers. — *end note*]

7.3.4 Concept `ConvertibleTo` [concepts.lib.corelang.convertibleto]

```
template <class T, class U>
concept bool ConvertibleTo =
  is_convertible<From, To>::value && // see below
  requires(From (&f)()) {
    static_cast<To>(f());
  };
```

1 Let `test` be the invented function:

```
To test(From (&f)()) {
  return f();
}
```

and let `f` be a function with no arguments and return type `From` such that `f()` is equality preserving. `ConvertibleTo<From, To>` is satisfied only if:

(1.1) — `To` is not an object or reference-to-object type, or `static_cast<To>(f())` is equal to `test(f)`.

(1.2) — `From` is not a reference-to-object type, or

(1.2.1) — If `From` is an rvalue reference to a non const-qualified type, the resulting state of the object referenced by `f()` after either above expression is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15).

(1.2.2) — Otherwise, the object referred to by `f()` is not modified by either above expression.

2 There need not be any subsumption relationship between `ConvertibleTo<From, To>` and `is_convertible<From, To>::value`.

7.3.5 Concept `CommonReference` [concepts.lib.corelang.commonref]

1 For two types `T` and `U`, if `common_reference_t<T, U>` is well-formed and denotes a type `C` such that both `ConvertibleTo<T, C>` and `ConvertibleTo<U, C>` are satisfied, then `T` and `U` share a *common reference type*, `C`. [*Note:* `C` could be the same as `T`, or `U`, or it could be a different type. `C` may be a reference type. `C` need not be unique. — *end note*]

```
template <class T, class U>
concept bool CommonReference =
  Same<common_reference_t<T, U>, common_reference_t<U, T>> &&
  ConvertibleTo<T, common_reference_t<T, U>> &&
  ConvertibleTo<U, common_reference_t<T, U>>;
```

2 Let `C` be `common_reference_t<T, U>`. Let `t` be a function whose return type is `T`, and let `u` be a function whose return type is `U`. `CommonReference<T, U>` is satisfied only if:

(2.1) — `C(t())` equals `C(t())` if and only if `t()` is an equality preserving expression (7.1.1).

(2.2) — `C(u())` equals `C(u())` if and only if `u()` is an equality preserving expression.

3 [*Note:* Users can customize the behavior of `CommonReference` by specializing the `basic_common_reference` class template (8.4.3). — *end note*]

7.3.6 Concept `Common` [concepts.lib.corelang.common]

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

```
template <class T, class U>
concept bool Common =
  Same<common_type_t<T, U>, common_type_t<U, T>> &&
  ConvertibleTo<T, common_type_t<T, U>> &&
  ConvertibleTo<U, common_type_t<T, U>> &&
  CommonReference<
    add_lvalue_reference_t<const T>,
    add_lvalue_reference_t<const U>> &&
  CommonReference<
    add_lvalue_reference_t<common_type_t<T, U>>,
    common_reference_t<
      add_lvalue_reference_t<const T>,
      add_lvalue_reference_t<const U>>>;
```

2 Let `C` be `common_type_t<T, U>`. Let `t` be a function whose return type is `T`, and let `u` be a function whose return type is `U`. `Common<T, U>` is satisfied only if:

(2.1) — `C(t())` equals `C(t())` if and only if `t()` is an equality preserving expression (7.1.1).

(2.2) — `C(u())` equals `C(u())` if and only if `u()` is an equality preserving expression (7.1.1).

3 [*Note:* Users can customize the behavior of `Common` by specializing the `common_type` class template (8.4.2). — *end note*]

7.3.7 Concept `Integral` [concepts.lib.corelang.integral]

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

1 There need not be any subsumption relationship between `Integral<T>` and `is_integral<T>::value`.

7.3.8 Concept `SignedIntegral` [concepts.lib.corelang.signedintegral]

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

1 There need not be any subsumption relationship between `SignedIntegral<T>` and `is_signed<T>::value`.

2 [*Note:* `SignedIntegral<T>` may be satisfied even for types that are not signed integral types (ISO/IEC 14882:2014 §3.9.1); for example, `char`. — *end note*]

7.3.9 Concept `UnsignedIntegral` [concepts.lib.corelang.unsignedintegral]

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

1 [*Note:* `UnsignedIntegral<T>` may be satisfied even for types that are not unsigned integral types (ISO/IEC 14882:2014 §3.9.1); for example, `char`. — *end note*]

7.3.10 Concept `Assignable` [concepts.lib.corelang.assignable]

```
template <class T, class U>
concept bool Assignable =
  is_lvalue_reference<T>::value && // see below
  CommonReference<
    const remove_reference_t<T>&,
    const remove_reference_t<U>&> &&
  requires(T t, U&& u) {
    { t = std::forward<U>(u) } -> Same<T>&&;
  };
```

1 Let `t` be an lvalue that refers to an object `o` such that `decltype((t))` is `T`, and `u` an expression such that `decltype((u))` is `U`. Let `u2` be a distinct object that is equal to `u`. `Assignable<T, U>` is satisfied only if

(1.1) — `addressof(t = u) == addressof(o)`.

(1.2) — After evaluating `t = u`:

(1.2.1) — `t` is equal to `u2`, unless `u` is a non-const xvalue that refers to `o`.

(1.2.2) — If `u` is a non-`const` xvalue, the resulting state of the object to which it refers is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15).

(1.2.3) — Otherwise, if `u` is a glvalue, the object to which it refers is not modified.

2 There need not be any subsumption relationship between `Assignable<T, U>` and `is_lvalue_reference<T>::value`.

3 [*Note:* Assignment need not be a total function (6.2.1.3); in particular, if assignment to an object `x` can result in a modification of some other object `y`, then `x = y` is likely not in the domain of `=`. — *end note*]

7.3.11 Concept `Swappable` [concepts.lib.corelang.swappable]

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

template <class T, class U>
```

```
concept bool SwappableWith =
  CommonReference<
    const remove_reference_t<T>&,
    const remove_reference_t<U>&> &&
  requires(T&& t, U&& u) {
    ranges::swap(std::forward<T>(t), std::forward<T>(t));
    ranges::swap(std::forward<U>(u), std::forward<U>(u));
    ranges::swap(std::forward<T>(t), std::forward<U>(u));
    ranges::swap(std::forward<U>(u), std::forward<T>(t));
  };
```

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 `SwappableWith<T, U>` is satisfied. `SwappableWith<T, U>` is satisfied only if given distinct objects `t2` equal to `t` and `u2` equal to `u`, after evaluating either `ranges::swap(t, u)` or `ranges::swap(u, t)`, `t2` is equal to `u` and `u2` is equal to `t`.

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

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

```
#include <utility>

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

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

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

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

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

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

— end example]

7.3.12 Concept `Destructible` [concepts.lib.corelang.destructible]

1 The `Destructible` concept specifies properties of all types, instances of which can be destroyed at the end of their lifetime, or reference types.

```
template <class T>
concept bool Destructible = is_nothrow_destructible<T>::value; // see below
```

2 There need not be any subsumption relationship between `Destructible<T>` and `is_nothrow_destructible<T>::value`.

3 [*Note:* Unlike the `Destructible` library concept in the C++ Standard (ISO/IEC 14882:2014 §17.6.3.1), this concept forbids destructors that are `noexcept(false)`, even if non-throwing. *— end note*]

7.3.13 Concept `Constructible` [concepts.lib.corelang.constructible]

1 The `Constructible` concept constrains the initialization of a variable of a type with a given set of argument types.

```
template <class T, class... Args>
concept bool Constructible =
  Destructible<T> && is_constructible<T, Args...>::value; // see below
```

2 There need not be any subsumption relationship between `Constructible<T, Args...>` and `is_constructible<T, Args...>::value`.

7.3.14 Concept `DefaultConstructible` [concepts.lib.corelang.defaultconstructible]

```
template <class T>
concept bool DefaultConstructible = Constructible<T>;
```

7.3.15 Concept `MoveConstructible` [concepts.lib.corelang.moveconstructible]

```
template <class T>
concept bool MoveConstructible =
  Constructible<T, T> && ConvertibleTo<T, T>;
```

1 If `T` is an object type, then let `rv` be an rvalue of type `T` and `u2` a distinct object of type `T` equal to `rv`. `MoveConstructible<T>` is satisfied only if

(1.1) — After the definition `T u = rv;`, `u` is equal to `u2`.

(1.2) — `T{rv}` is equal to `u2`.

(1.3) — If `T` is not `const`, `rv`'s resulting state is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15); otherwise, it is unchanged.

7.3.16 Concept `CopyConstructible` [concepts.lib.corelang.copyconstructible]

```
template <class T>
concept bool CopyConstructible =
  MoveConstructible<T> &&
  Constructible<T, T&> && ConvertibleTo<T&, T> &&
  Constructible<T, const T&> && ConvertibleTo<const T&, T> &&
  Constructible<T, const T> && ConvertibleTo<const T, T>;
```

1 If `T` is an object type, then let `v` be an lvalue of type (possibly `const`) `T` or an rvalue of type `const T`. `CopyConstructible<T>` is satisfied only if

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

(1.2) — `T{v}` is equal to `v`.

7.4 Comparison concepts [concepts.lib.compare]

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

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

7.4.2 Concept `Boolean` [concepts.lib.compare.boolean]

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

```
template <class B>
concept bool Boolean =
  Movable<decay_t<B>> && // (see 7.5.1)
  requires(const remove_reference_t<B>& b1,
           const remove_reference_t<B>& b2, const bool a) {
    { b1 }       -> ConvertibleTo<bool>&&;
    { !b1 }      -> ConvertibleTo<bool>&&;
    { b1 && a }  -> Same<bool>&&;
    { b1 || a }  -> Same<bool>&&;
    { b1 && b2 } -> Same<bool>&&;
    { a && b2  } -> Same<bool>&&;
    { b1 || b2 } -> Same<bool>&&;
    { a || b2  } -> Same<bool>&&;
    { b1 == b2 } -> ConvertibleTo<bool>&&;
    { b1 == a  } -> ConvertibleTo<bool>&&;
    { a == b2  } -> ConvertibleTo<bool>&&;
    { b1 != b2 } -> ConvertibleTo<bool>&&;
    { b1 != a  } -> ConvertibleTo<bool>&&;
    { a != b2  } -> ConvertibleTo<bool>&&;
  };
```

2 Given `const` lvalues `b1` and `b2` of type `remove_reference_t<B>`, then `Boolean<B>` is satisfied only if

(2.1) — `bool(b1) == !bool(!b1)`.

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

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

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

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

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

7.4.3 Concept `EqualityComparable` [concepts.lib.compare.equalitycomparable]

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

1 Let `t` and `u` be `const` lvalues of types `remove_reference_t<T>` and `remove_reference_t<U>` respectively. `WeaklyEqualityComparableWith<T, U>` is satisfied only if:

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

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

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

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

```
template <class T>
concept bool EqualityComparable = WeaklyEqualityComparableWith<T, T>;
```

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

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

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

```
template <class T, class U>
concept bool EqualityComparableWith =
  EqualityComparable<T> &&
  EqualityComparable<U> &&
  CommonReference<
    const remove_reference_t<T>&,
    const remove_reference_t<U>&> &&
  EqualityComparable<
    common_reference_t<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&>> &&
  WeaklyEqualityComparableWith<T, U>;
```

4 Let `t` be a `const` lvalue of type `remove_reference_t<T>`, `u` be a `const` lvalue of type `remove_reference_t<U>`, and `C` be `common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`. `EqualityComparableWith<T, U>` is satisfied only if:

(4.1) — `bool(t == u) == bool(C(t) == C(u))`.

7.4.4 Concept `StrictTotallyOrdered` [concepts.lib.compare.stricttotallyordered]

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

```
    { a <= b } -> Boolean&&;
    { a >= b } -> Boolean&&;
  };
```

1 Let `a`, `b`, and `c` be `const` lvalues of type `remove_reference_t<T>`. `StrictTotallyOrdered<T>` is satisfied only if

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

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

(1.3) — `bool(a > b) == bool(b < a)`.

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

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

```
template <class T, class U>
concept bool StrictTotallyOrderedWith =
  StrictTotallyOrdered<T> &&
  StrictTotallyOrdered<U> &&
  CommonReference<
    const remove_reference_t<T>&,
    const remove_reference_t<U>&> &&
  StrictTotallyOrdered<
    common_reference_t<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&>> &&
  EqualityComparableWith<T, U> &&
  requires(const remove_reference_t<T>& t,
           const remove_reference_t<U>& u) {
    { t < u }  -> Boolean&&;
    { t > u }  -> Boolean&&;
    { t <= u } -> Boolean&&;
    { t >= u } -> Boolean&&;
    { u < t }  -> Boolean&&;
    { u > t }  -> Boolean&&;
    { u <= t } -> Boolean&&;
    { u >= t } -> Boolean&&;
  };
```

2 Let `t` be a `const` lvalue of type `remove_reference_t<T>`, `u` be a `const` lvalue of type `remove_reference_t<U>`, and `C` be `common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`. `StrictTotallyOrderedWith<T, U>` is satisfied only if

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

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

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

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

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

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

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

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

7.5 Object concepts [concepts.lib.object]

1 This section describes concepts that specify the basis of the value-oriented programming style on which the library is based.

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

```
template <class T>
concept bool Movable =
  is_object<T>::value &&
  MoveConstructible<T> &&
  Assignable<T&, T> &&
  Swappable<T>;
```

1 There need not be any subsumption relationship between `Movable<T>` and `is_object<T>::value`.

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

```
template <class T>
concept bool Copyable =
  CopyConstructible<T> &&
  Movable<T> &&
  Assignable<T&, const T&>;
```

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

```
template <class T>
concept bool Semiregular =
  Copyable<T> &&
  DefaultConstructible<T>;
```

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

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

```
template <class T>
concept bool Regular =
  Semiregular<T> &&
  EqualityComparable<T>;
```

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

7.6 Callable concepts [concepts.lib.callable]

7.6.1 In general [concepts.lib.callable.general]

1 The concepts in this section describe the requirements on function objects (8.3) and their arguments.

7.6.2 Concept Invocable [concepts.lib.callable.invocable]

1 The `Invocable` concept specifies a relationship between a callable type (ISO/IEC 14882:2014 §20.9.1) `F` and a set of argument types `Args...` which can be evaluated by the library function `invoke` (8.3.1).

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

2 [*Note:* Since the `invoke` function call expression is not required to be equality-preserving (7.1.1), a function that generates random numbers may satisfy `Invocable`. — *end note*]

7.6.3 Concept `RegularInvocable` [concepts.lib.callable.regularinvocable]

```
template <class F, class... Args>
concept bool RegularInvocable =
  Invocable<F, Args...>;
```

1 The `invoke` function call expression shall be equality-preserving and shall not modify the function object or the arguments (7.1.1). [*Note:* This requirement supersedes the annotation in the definition of `Invocable`. — *end note*]

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

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

7.6.4 Concept `Predicate` [concepts.lib.callable.predicate]

```
template <class F, class... Args>
concept bool Predicate =
  RegularInvocable<F, Args...> &&
  Boolean<result_of_t<F&&(Args&&...)>>;
```

7.6.5 Concept `Relation` [concepts.lib.callable.relation]

```
template <class R, class T, class U>
concept bool Relation =
  Predicate<R, T, T> &&
  Predicate<R, U, U> &&
  CommonReference<
    const remove_reference_t<T>&,
    const remove_reference_t<U>&> &&
  Predicate<R,
    common_reference_t<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&>,
    common_reference_t<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&>> &&
  Predicate<R, T, U> &&
  Predicate<R, U, T>;
```

1 Let `r` be an expression such that `decltype((r))` is `R`, `t` be an expression such that `decltype((t))` is `T`, `u` be an expression such that `decltype((u))` is `U`, and `C` be `common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>`. `Relation<R, T, U>` is satisfied only if

(1.1) — `bool(r(t, u)) == bool(r(C(t), C(u)))`.

(1.2) — `bool(r(u, t)) == bool(r(C(u), C(t)))`.

7.6.6 Concept `StrictWeakOrder` [concepts.lib.callable.strictweakorder]

```
template <class R, class T, class U>
concept bool StrictWeakOrder = Relation<R, T, U>;
```

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

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

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

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

(2.2.1) — `equiv` is an equivalence relation

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

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

8 General utilities library [utilities]

8.1 General [utilities.general]

1 This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the Ranges library. These utilities are summarized in Table 4.

Table 4 — General utilities library summary

Subclause		Header(s)
8.2	Utility components	`<experimental/ranges/utility>`
8.3	Function objects	`<experimental/ranges/functional>`
8.4	Type traits	`<type_traits>`
8.5	Tagged tuple-like types	`<experimental/ranges/utility>` & `<experimental/ranges/tuple>`

8.2 Utility components [utility]

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

Header `<experimental/ranges/utility>` synopsis

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

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 8.2.1, swap:
  namespace {
    constexpr unspecified  swap = unspecified ;
  }

  // 8.2.2, exchange:
  template <MoveConstructible T, class U=T>
    requires Assignable<T&, U>
  constexpr T exchange(T& obj, U&& new_val) noexcept(see below);

  // 8.5.2, struct with named accessors
  template <class T>
  concept bool TagSpecifier = see below;

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

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

  // 8.5.4, tagged pairs
  template <TaggedType T1, TaggedType T2> using tagged_pair = see below;
```

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

namespace std {
  // 8.5.3, tuple-like access to tagged
  template <class Base, class... Tags>
  struct tuple_size<experimental::ranges::tagged<Base, Tags...>>;

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

8.2.1 swap [utility.swap]

1 The name `swap` denotes a customization point object (6.3.5.1). The effect of the expression `ranges::swap(E1, E2)` for some expressions `E1` and `E2` is equivalent to:

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

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

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

(1.2) — Otherwise, `(void)swap_ranges(E1, E2)` if `E1` and `E2` are lvalues of array types (ISO/IEC 14882:2014 §3.9.2) of equal extent and `ranges::swap(*(E1), *(E2))` is a valid expression, except that `noexcept(ranges::swap(E1, E2))` is equal to `noexcept(ranges::swap(*(E1), *(E2)))`.

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

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

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

8.2.2 exchange [utility.exchange]

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

1 *Effects:* Equivalent to:

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

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

```
is_nothrow_move_constructible<T>::value &&
is_nothrow_assignable<T&, U>::value
```

8.3 Function objects [function.objects]

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

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 8.3.1, invoke:
  template <class F, class... Args>
  result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

  // 8.3.2, comparisons:
  template <class T = void>
    requires see below
  struct equal_to;

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

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

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

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

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

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

  // 8.3.3, identity:
  struct identity;
}}}}
```

8.3.1 Function template `invoke` [func.invoke]

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

1 *Effects:* Equivalent to:
`return INVOKE(std::forward<F>(f), std::forward<Args>(args)...);` (ISO/IEC 14882:2014 § 20.9.2).

8.3.2 Comparisons [comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language (ISO/IEC 14882:2014 §5.9, ISO/IEC 14882:2014 §5.10).

2 In this section, *`BUILTIN_PTR_CMP`*`(T,` *op*`, U)` for types `T` and `U` and where *op* is an equality (ISO/IEC 14882:2014 §5.10) or relational operator (ISO/IEC 14882:2014 §5.9) is a boolean constant expression. *`BUILTIN_PTR_CMP`*`(T,` *op*`, U)` is `true` if and only if *op* in the expression `declval<T>()` *op* `declval<U>()` resolves to a built-in operator comparing pointers.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  typedef unspecified  is_transparent;
};
```

10 *Requires:* If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator `==` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving (7.1.1).

11 *Effects:*

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

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

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

  typedef unspecified  is_transparent;
};
```

12 `operator()` has effects equivalent to:

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

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

  typedef unspecified  is_transparent;
};
```

13 `operator()` has effects equivalent to:

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

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

  typedef unspecified  is_transparent;
};
```

14 *Requires:* If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator `<` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving (7.1.1). For any expressions `ET` and `EU` such that `decltype((ET))` is `T` and `decltype((EU))` is `U`, exactly one of `less<>{}(ET, EU)`, `less<>{}(EU, ET)` or `equal_to<>{}(ET, EU)` shall be `true`.

15 *Effects:*

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

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

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

  typedef unspecified  is_transparent;
};
```

16 `operator()` has effects equivalent to:

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

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

  typedef unspecified  is_transparent;
};
```

17 `operator()` has effects equivalent to:

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

8.3.3 Class identity [func.identity]

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

  typedef unspecified  is_transparent;
};
```

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

8.4 Metaprogramming and type traits [meta]

8.4.1 Header <experimental/ranges/type_traits> synopsis [meta.type.synop]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 8.4.2, type properties:
  template <class T, class U> struct is_swappable_with;
  template <class T> struct is_swappable;
```

```
  template <class T, class U> struct is_nothrow_swappable_with;
  template <class T> struct is_nothrow_swappable;

  template <class T, class U> constexpr bool is_swappable_with_v
    = is_swappable_with<T, U>::value;
  template <class T> constexpr bool is_swappable_v
    = is_swappable<T>::value;

  template <class T, class U> constexpr bool is_nothrow_swappable_with_v
    = is_nothrow_swappable_with<T, U>::value;
  template <class T> constexpr bool is_nothrow_swappable_v
    = is_nothrow_swappable<T>::value;

  // 8.4.3, other transformations:
  template <class... T> struct common_type;
  template <class T, class U, template <class> class TQual, template <class> class UQual>
    struct basic_common_reference { };
  template <class... T> struct common_reference;

  template <class... T>
    using common_type_t = typename common_type<T...>::type;
  template <class... T>
    using common_reference_t = typename common_reference<T...>::type;
}}}}
```

8.4.2 Type properties [meta.unary.prop]

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial specializations of any of these templates.

3 For all of the class templates `X` declared in this subclause, instantiating that template with a template argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of `X` require that the argument must be a complete type.

4 For the purpose of defining the templates in this subclause, a function call expression `declval<T>()` for any type `T` is considered to be a trivial (ISO/IEC 14882:2014 §3.9, ISO/IEC 14882:2014 §12) function call that is not an odr-use (ISO/IEC 14882:2014 §3.2) of `declval` in the context of the corresponding definition notwithstanding the restrictions of (ISO/IEC 14882:2014 §20.2.5).

Table 5 — Additional type property predicates

Template	Condition	Precondition
`template <class T, class U>` `struct is_swappable_with;`	The expressions `ranges::swap(declval<T>(), declval<U>())` and `ranges::swap(declval<U>(), declval<T>())` are each well-formed when treated as an unevaluated operand (Clause ISO/IEC 14882:2014 §5). Access checking is performed as if in a context unrelated to `T` and `U`. Only the validity of the immediate context of the `swap` expressions is considered. [*Note:* The compilation of the expressions can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]	`T` and `U` shall be complete types, (possibly *cv*-qualified) `void`, or arrays of unknown bound.
`template <class T>` `struct is_swappable;`	For a referenceable type `T`, the same result as `is_swappable_with_v<T&,T&>`, otherwise `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.
`template <class T, class U>` `struct is_nothrow_swappable_with;`	`is_swappable_with_v<T, U>` is `true` and each `swap` expression of the definition of `is_swappable_with<T, U>` is known not to throw any exceptions (ISO/IEC 14882:2014 §5.3.7).	`T` and `U` shall be complete types, (possibly *cv*-qualified) `void`, or arrays of unknown bound.

Table 5 — Additional type property predicates (continued)

Template	Condition	Precondition
`template <class T>` `struct is_nothrow_swappable;`	For a referenceable type `T`, the same result as `is_nothrow_swappable_with_v<T&, T&>`, otherwise `false`.	`T` shall be a complete type, (possibly *cv*-qualified) `void`, or an array of unknown bound.

8.4.3 Other transformations [meta.trans.other]

Table 6 — Other transformations

Template	Comments
`template <class... T>` `struct common_type;`	The member typedef `type` shall be defined or omitted as specified below. If it is omitted, there shall be no member `type`. Each type in the parameter pack `T` shall be complete or (possibly *cv*) `void`. A program may specialize this trait if at least one template parameter in the specialization depends on a user-defined type and `sizeof...(T) == 2`. [*Note:* Such specializations are needed when only explicit conversions are desired among the template arguments. — *end note*]
`template <class T, class U,` `template <class> class TQual,` `template <class> class UQual>` `struct basic_common_reference;`	The primary template shall have no member typedef `type`. A program may specialize this trait if at least one template parameter in the specialization depends on a user-defined type. In such a specialization, a member typedef `type` may be defined or omitted. If it is omitted, there shall be no member `type`. [*Note:* Such specializations may be used to influence the result of `common_reference`. — *end note*]
`template <class... T>` `struct common_reference;`	The member typedef `type` shall be defined or omitted as specified below. If it is omitted, there shall be no member `type`. Each type in the parameter pack `T` shall be complete or (possibly *cv*) `void`.

1 Let `CREF(A)` be `add_lvalue_reference_t<const remove_reference_t<A>>`. Let `UNCVREF(A)` be `remove_cv_t<remove_reference_t<A>>`. Let `XREF(A)` denote a unary template `T` such that `T<UNCVREF(A)>` denotes the same type as `A`. Let `COPYCV(FROM, TO)` be an alias for type `TO` with the addition of `FROM`'s top-level *cv*-qualifiers. [*Example:* `COPYCV(const int, volatile short)` is an alias for `const volatile short`. — *end example*] Let `RREF_RES(Z)` be `remove_reference_t<Z>&&` if `Z` is a reference type or `Z` otherwise. Let `COND_RES(X, Y)` be `decltype(declval<bool>() ? declval<X(&)()>()() : declval<Y(&)()>()())`. Given types `A` and `B`, let `X` be `remove_reference_t<A>`, let `Y` be `remove_reference_t<B>`, and let `COMMON_REF(A, B)` be:

(1.1) — If `A` and `B` are both lvalue reference types, `COMMON_REF(A, B)` is `COND_RES(COPYCV(X, Y) &, COPYCV(Y, X) &)`.

(1.2) — Otherwise, let `C` be `RREF_RES(COMMON_REF(X&, Y&))`. If `A` and `B` are both rvalue reference types, and `C` is well-formed, and `is_convertible<A, C>::value` and `is_convertible<B, C>::value` are `true`, then `COMMON_REF(A, B)` is `C`.

(1.3) — Otherwise, let `D` be `COMMON_REF(const X&, Y&)`. If `A` is an rvalue reference and `B` is an lvalue reference and `D` is well-formed and `is_convertible<A, D>::value` is `true`, then `COMMON_REF(A, B)` is `D`.

(1.4) — Otherwise, if `A` is an lvalue reference and `B` is an rvalue reference, then `COMMON_REF(A, B)` is `COMMON_REF(B, A)`.

(1.5) — Otherwise, `COMMON_REF(A, B)` is `decay_t<COND_RES(CREF(A), CREF(B))>`.

If any of the types computed above are ill-formed, then `COMMON_REF(A, B)` is ill-formed.

2 Note A: For the `common_type` trait applied to a parameter pack `T` of types, the member `type` shall be either defined or not present as follows:

(2.1) — If `sizeof...(T)` is zero, there shall be no member `type`.

(2.2) — Otherwise, if `sizeof...(T)` is one, let T1 denote the sole type in the pack `T`. The member typedef `type` shall denote the same type as `decay_t<T1>`.

(2.3) — Otherwise, if `sizeof...(T)` is two, let `T1` and `T2` denote the two types in the pack `T`, and let `D1` and `D2` be `decay_t<T1>` and `decay_t<T2>` respectively. Then

(2.3.1) — If `D1` and `T1` denote the same type and `D2` and `T2` denote the same type, then

(2.3.1.1) — If `std::common_type_t<T1, T2>` is well-formed, then the member typedef `type` denotes `std::common_type_t<T1, T2>`.

(2.3.1.2) — If `COMMON_REF(T1, T2)` is well-formed, then the member typedef `type` denotes that type.

(2.3.1.3) — Otherwise, there shall be no member `type`.

(2.3.2) — Otherwise, if `common_type_t<D1, D2>` is well-formed, then the member typedef `type` denotes that type.

(2.3.3) — Otherwise, there shall be no member `type`.

(2.4) — Otherwise, if `sizeof...(T)` is greater than two, let `T1`, `T2`, and `Rest`, respectively, denote the first, second, and (pack of) remaining types comprising `T`. Let `C` be the type `common_type_t<T1, T2>`. Then:

(2.4.1) — If there is such a type `C`, the member typedef `type` shall denote the same type, if any, as `common_type_t<C, Rest...>`.

(2.4.2) — Otherwise, there shall be no member `type`.

3 Note B: Notwithstanding the provisions of ISO/IEC 14882:2014 §20.10.2, and pursuant to ISO/IEC 14882:2014 §17.6.4.2.1, a program may specialize `common_type<T1, T2>` for types `T1` and `T2` such that `is_same<T1, decay_t<T1>>::value` and `is_same<T2, decay_t<T2>>::value` are each `true`. [*Note:* Such specializations are needed when only explicit conversions are desired between the template arguments. — *end note*] Such a specialization need not have a member named `type`, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous *cv*-unqualified non-reference type `C` to which each of the types `T1` and `T2` is explicitly convertible. Moreover, `common_type_t<T1, T2>` shall denote the same type, if any, as does `common_type_t<T2, T1>`. No diagnostic is required for a violation of this Note's rules.

4 For the `common_reference` trait applied to a parameter pack `T` of types, the member `type` shall be either defined or not present as follows:

(4.1) — If `sizeof...(T)` is zero, there shall be no member `type`.

(4.2) — Otherwise, if `sizeof...(T)` is one, let `T1` denote the sole type in the pack `T`. The member typedef `type` shall denote the same type as `T1`.

(4.3) — Otherwise, if `sizeof...(T)` is two, let `T1` and `T2` denote the two types in the pack `T`. Then

(4.3.1) — If `T1` and `T2` are reference types and `COMMON_REF(T1, T2)` is well-formed and denotes a reference type then the member typedef `type` denotes that type.

(4.3.2) — Otherwise, if `basic_common_reference<UNCVREF(T1), UNCVREF(T2), XREF(T1), XREF(T2)>::type` is well-formed, then the member typedef `type` denotes that type.

(4.3.3) — Otherwise, if `COND_RES(T1, T2)` is well-formed, then the member typedef `type` denotes that type.

(4.3.4) — Otherwise, if `common_type_t<T1, T2>` is well-formed, then the member typedef `type` denotes that type.

(4.3.5) — Otherwise, there shall be no member `type`.

(4.4) — Otherwise, if `sizeof...(T)` is greater than two, let `T1`, `T2`, and `Rest`, respectively, denote the first, second, and (pack of) remaining types comprising `T`. Let `C` be the type `common_reference_t<T1, T2>`. Then:

(4.4.1) — If there is such a type `C`, the member typedef `type` shall denote the same type, if any, as `common_reference_t<C, Rest...>`.

(4.4.2) — Otherwise, there shall be no member `type`.

5 Notwithstanding the provisions of ISO/IEC 14882:2014 §20.10.2, and pursuant to ISO/IEC 14882:2014 § 17.6.4.2.1, a program may specialize `basic_common_reference<T, U, TQual, UQual>` for types `T` and `U` such that `is_same<T, decay_t<T>>::value` and `is_same<U, decay_t<U>>::value` are each `true`. [*Note:* Such specializations are needed when only explicit conversions are desired between the template arguments. — *end note*] Such a specialization need not have a member named `type`, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous type `C` to which each of the types `TQual<T>` and `UQual<U>` is convertible. Moreover, `basic_common_reference<T, U, TQual, UQual>::type` shall denote the same type, if any, as does `basic_common_reference<U, T, UQual, TQual>::type`. A program may not specialize `basic_common_reference` on the third or fourth parameters, `TQual` or `UQual`. No diagnostic is required for a violation of these rules.

8.5 Tagged tuple-like types [taggedtup]

8.5.1 In general [taggedtup.general]

1 The library provides a template for augmenting a tuple-like type with named element accessor member functions. The library also provides several templates that provide access to `tagged` objects as if they were `tuple` objects (see ISO/IEC 14882:2014 §20.4.2.6).

8.5.2 Class template `tagged` [taggedtup.tagged]

1 Class template `tagged` augments a tuple-like class type (e.g., `pair` (ISO/IEC 14882:2014 §20.3), `tuple` (ISO/IEC 14882:2014 §20.4)) by giving it named accessors. It is used to define the alias templates `tagged_pair` (8.5.4) and `tagged_tuple` (8.5.5).

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

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

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

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

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

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

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

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

5 A *tag specifier* is a type that facilitates a mapping from a tuple-like type and an element index into a *tagged getter* that gives named access to the element at that index. `TagSpecifier<T>` is satisfied if and only if

T is a tag specifier. The tag specifiers in the Tags parameter pack shall be unique. [*Note:* The mapping mechanism from tag specifier to tagged getter is unspecified. — *end note*]

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

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

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

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

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

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

```
is_nothrow_move_constructible<Base>::value
```

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

11 *Effects:* Initializes Base with that.

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

```
is_nothrow_copy_constructible<Base>::value
```

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

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

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

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

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

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

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

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

17 *Returns:* *this.

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

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

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

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

20 *Returns:* `*this`.

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

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

22 *Returns:* `*this`.

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

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

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

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

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

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

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

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

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

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

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

8.5.3 Tuple-like access to `tagged` [tagged.astuple]

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

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

8.5.4 Alias template `tagged_pair` [tagged.pairs]

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

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

1 [*Example:*

```
// See 11.2:
tagged_pair<tag::min(int), tag::max(int)> p{0, 1};
assert(&p.min() == &p.first);
assert(&p.max() == &p.second);
```

— *end example*]

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

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

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

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

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

3 [*Example:* In place of:

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

a C++ program may contain:

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

— *end example*]

8.5.5 Alias template tagged_tuple [tagged.tuple]

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

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

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

2

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

3 [*Example:*

```
// See 11.2:
tagged_tuple<tag::in(char*), tag::out(char*)> t{0, 0};
assert(&t.in() == &get<0>(t));
assert(&t.out() == &get<1>(t));
```

— *end example*]

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

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

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

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

3 [*Example:*

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

creates a tagged tuple of type

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

— *end example*]

9 Iterators library [iterators]

9.1 General [iterators.general]

1 This Clause describes components that C++ programs may use to perform iterations over containers (Clause ISO/IEC 14882:2014 §23), streams (ISO/IEC 14882:2014 §27.7), and stream buffers (ISO/IEC 14882:2014 §27.6).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 7.

Table 7 — Iterators library summary

	Subclause	Header(s)
9.3	Iterator requirements	
9.4	Indirect callable requirements	
9.5	Common algorithm requirements	
9.6	Iterator primitives	`<experimental/ranges/iterator>`
9.7	Predefined iterators	
9.8	Stream iterators	

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

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

  // 9.3, iterator requirements:
  // 9.3.2, customization points:
  namespace {
    // 9.3.2.1, iter_move:
    constexpr unspecified  iter_move = unspecified ;

    // 9.3.2.2, iter_swap:
    constexpr unspecified  iter_swap = unspecified ;
  }

  // 9.3.3, associated types:
  // 9.3.3.1, difference_type:
  template <class> struct difference_type;
  template <class T> using difference_type_t
    = typename difference_type<T>::type;

  // 9.3.3.2, value_type:
  template <class> struct value_type;
  template <class T> using value_type_t
    = typename value_type<T>::type;

  // 9.3.3.3, iterator_category:
  template <class> struct iterator_category;
  template <class T> using iterator_category_t
```

```
  = typename iterator_category<T>::type;

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

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

// 9.3.4, Readable:
template <class In>
concept bool Readable = see below;

// 9.3.5, Writable:
template <class Out, class T>
concept bool Writable = see below;

// 9.3.6, WeaklyIncrementable:
template <class I>
concept bool WeaklyIncrementable = see below;

// 9.3.7, Incrementable:
template <class I>
concept bool Incrementable = see below;

// 9.3.8, Iterator:
template <class I>
concept bool Iterator = see below;

// 9.3.9, Sentinel:
template <class S, class I>
concept bool Sentinel = see below;

// 9.3.10, SizedSentinel:
template <class S, class I>
  constexpr bool disable_sized_sentinel = false;

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

// 9.3.11, InputIterator:
template <class I>
concept bool InputIterator = see below;

// 9.3.12, OutputIterator:
template <class I>
concept bool OutputIterator = see below;

// 9.3.13, ForwardIterator:
template <class I>
concept bool ForwardIterator = see below;

// 9.3.14, BidirectionalIterator:
template <class I>
concept bool BidirectionalIterator = see below;
```

```
// 9.3.15, RandomAccessIterator:
template <class I>
concept bool RandomAccessIterator = see below;

// 9.4, indirect callable requirements:
// 9.4.2, indirect callables:
template <class F, class I>
concept bool IndirectUnaryInvocable = see below;

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

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

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

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

template <class> struct indirect_result_of;

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

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

// 9.4.3, projected:
template <Readable I, IndirectRegularUnaryInvocable<I> Proj>
struct projected;

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

// 9.5, common algorithm requirements:
// 9.5.2 IndirectlyMovable:
template <class In, class Out>
concept bool IndirectlyMovable = see below;

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

// 9.5.3 IndirectlyCopyable:
template <class In, class Out>
concept bool IndirectlyCopyable = see below;

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

// 9.5.4 IndirectlySwappable:
```

```
template <class I1, class I2 = I1>
concept bool IndirectlySwappable = see below;

// 9.5.5 IndirectlyComparable:
template <class I1, class I2, class R = equal_to<>, class P1 = identity,
    class P2 = identity>
concept bool IndirectlyComparable = see below;

// 9.5.6 Permutable:
template <class I>
concept bool Permutable = see below;

// 9.5.7 Mergeable:
template <class I1, class I2, class Out,
    class R = less<>, class P1 = identity, class P2 = identity>
concept bool Mergeable = see below;

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

// 9.6, primitives:
// 9.6.1, traits:
template <class Iterator> using iterator_traits = see below;

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

// 9.6.3, iterator tags:
struct output_iterator_tag { };
struct input_iterator_tag { };
struct forward_iterator_tag : input_iterator_tag { };
struct bidirectional_iterator_tag : forward_iterator_tag { };
struct random_access_iterator_tag : bidirectional_iterator_tag { };

// 9.6.4, iterator operations:
namespace {
  constexpr unspecified  advance = unspecified ;
  constexpr unspecified  distance = unspecified ;
  constexpr unspecified  next = unspecified ;
  constexpr unspecified  prev = unspecified ;
}

// 9.7, predefined iterators and sentinels:

// 9.7.1, reverse iterators:
template <BidirectionalIterator I> class reverse_iterator;

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

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

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

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

  // 9.7.2, insert iterators:
  template <class Container> class back_insert_iterator;
  template <class Container>
    back_insert_iterator<Container> back_inserter(Container& x);

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

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

  // 9.7.3, move iterators and sentinels:
  template <InputIterator I> class move_iterator;
  template <class I1, class I2>
      requires EqualityComparableWith<I1, I2>
    constexpr bool operator==(
      const move_iterator<I1>& x, const move_iterator<I2>& y);
```

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

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

template <Semiregular S> class move_sentinel;

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

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

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

// 9.7.4, common iterators:
template <Iterator I, Sentinel<I> S>
  requires !Same<I, S>
class common_iterator;

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

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

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

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

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

// 9.7.5, default sentinels:
class default_sentinel;

// 9.7.6, counted iterators:
template <Iterator I> class counted_iterator;

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

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

// 9.7.8, unreachable sentinels:
class unreachable;
template <Iterator I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator==(unreachable, const I&) noexcept;
template <Iterator I>
  constexpr bool operator!=(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator!=(unreachable, const I&) noexcept;

// 9.7.7, dangling wrapper:
template <class T> class dangling;

// 9.8, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
          const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator==(default_sentinel x,
          const istream_iterator<T, charT, traits, Distance>& y);
```

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

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

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

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

namespace std {
  // 9.6.2, iterator traits:
  template <experimental::ranges::Iterator Out>
    struct iterator_traits<Out>;
  template <experimental::ranges::InputIterator In>
    struct iterator_traits<In>;
  template <experimental::ranges::InputIterator In>
      requires experimental::ranges::Sentinel<In, In>
    struct iterator_traits;
}
```

9.3 Iterator requirements [iterator.requirements]

9.3.1 In general [iterator.requirements.general]

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

2 Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This document defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 8.

Table 8 — Relations among iterator categories

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

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

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

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

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

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

8 An iterator and a sentinel denoting a range are comparable. The types of a sentinel and an iterator that

denote a range must satisfy `Sentinel` (9.3.9). A range `[i,s)` is empty if `i == s`; otherwise, `[i,s)` refers to the elements in the data structure starting with the element pointed to by `i` and up to but not including the element pointed to by the first iterator `j` such that `j == s`.

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

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

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

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

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

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

9.3.2 Customization points [iterator.custpoints]

9.3.2.1 `iter_move` [iterator.custpoints.iter_move]

1 The name `iter_move` denotes a *customization point object* (6.3.5.1). The expression `ranges::iter_move(E)` for some subexpression `E` is expression-equivalent to the following:

(1.1) — `static_cast<decltype(iter_move(E))>(iter_move(E))`, if that expression is well-formed when evaluated in a context that does not include `ranges::iter_move` but does include the lookup set produced by argument-dependent lookup (ISO/IEC 14882:2014 §3.4.2).

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

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

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

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

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

9.3.2.2 `iter_swap` [iterator.custpoints.iter_swap]

1 The name `iter_swap` denotes a *customization point object* (6.3.5.1). The expression `ranges::iter_swap(E1, E2)` for some subexpressions `E1` and `E2` is expression-equivalent to the following:

(1.1) — `(void)iter_swap(E1, E2)`, if that expression is well-formed when evaluated in a context that does not include `ranges::iter_swap` but does include the lookup set produced by argument-dependent lookup (ISO/IEC 14882:2014 §3.4.2) and the following declaration:

```
void iter_swap(auto, auto) = delete;
```

(1.2) — Otherwise, if the types of `E1` and `E2` both satisfy `Readable`, and if the reference type of `E1` is swappable with (7.3.11) the reference type of `E2`, then `ranges::swap(*E1, *E2)`

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

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

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

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

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

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

4 *Effects:* Equivalent to:

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

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

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

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

```
is_nothrow_constructible<value_type_t<T1>, rvalue_reference_t<T1>>::value &&
is_nothrow_assignable<value_type_t<T1>&, rvalue_reference_t<T1>>::value &&
is_nothrow_assignable<reference_t<T1>, rvalue_reference_t<T2>>::value &&
is_nothrow_assignable<reference_t<T1>, value_type_t<T2>>::value> &&
is_nothrow_move_constructible<value_type_t<T1>>::value &&
noexcept(ranges::iter_move(declval<T1&>()))
```

9.3.3 Iterator associated types [iterator.assoc.types]

1 To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if `WI` is the name of a type that satisfies the `WeaklyIncrementable` concept (9.3.6), `R` is the name of a type that satisfies the `Readable` concept (9.3.4), and `II` is the name of a type that satisfies the `InputIterator` concept (9.3.11) concept, the types

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

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

9.3.3.1 `difference_type` [iterator.assoc.types.difference_type]

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

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

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

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

template <class T>
  requires requires { typename T::difference_type; }
```

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

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

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

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

9.3.3.2 `value_type` [iterator.assoc.types.value_type]

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

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

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

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

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

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

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

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

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

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

4 When instantiated with a type `I` such that `I::value_type` is valid and denotes a type, `value_type<I>::type` names that type, unless it is not an object type (ISO/IEC 14882:2014 §3.9) in which case `value_type<I>`

shall have no nested type `type`. [*Note:* Some legacy output iterators define a nested type named `value_type` that is an alias for `void`. These types are not `Readable` and have no associated value types. — *end note*]

5 When instantiated with a type `I` such that `I::element_type` is valid and denotes a type, `value_type<I>::type` names the type `remove_cv_t<I::element_type>`, unless it is not an object type (ISO/IEC 14882:2014 §3.9) in which case `value_type<I>` shall have no nested type `type`. [*Note:* Smart pointers like `shared_ptr<int>` are `Readable` and have an associated value type. But a smart pointer like `shared_ptr<void>` is not `Readable` and has no associated value type. — *end note*]

9.3.3.3 `iterator_category` [iterator.assoc.types.iterator_category]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.3.4 Concept `Readable` [iterators.readable]

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

```
template <class In>
concept bool Readable =
  requires {
    typename value_type_t<In>;
    typename reference_t<In>;
    typename rvalue_reference_t<In>;
  } &&
  CommonReference<reference_t<In>&&, value_type_t<In>&> &&
  CommonReference<reference_t<In>&&, rvalue_reference_t<In>&&> &&
  CommonReference<rvalue_reference_t<In>&&, const value_type_t<In>&>;
```

9.3.5 Concept `Writable` [iterators.writable]

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

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

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

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

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

4 If `E` is an xvalue (ISO/IEC 14882:2014 §3.10), the resulting state of the object it denotes is valid but unspecified (ISO/IEC 14882:2014 §17.6.5.15).

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*]

9.3.6 Concept `WeaklyIncrementable` [iterators.weaklyincrementable]

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

```
template <class I>
concept bool WeaklyIncrementable =
  Semiregular<I> &&
  requires(I i) {
```

```
    typename difference_type_t<I>;
    requires SignedIntegral<difference_type_t<I>>;
    { ++i } -> Same<I>&; // not required to be equality preserving
    i++; // not required to be equality preserving
  };
```

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

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

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

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

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

9.3.7 Concept `Incrementable` [iterators.incrementable]

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

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

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

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

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

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

9.3.8 Concept `Iterator` [iterators.iterator]

1 The `Iterator` concept forms the basis of the iterator concept taxonomy; every iterator satisfies the `Iterator` requirements. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (9.3.9), to read (9.3.11) or write (9.3.12) values, or to provide a richer set of iterator movements (9.3.13, 9.3.14, 9.3.15).)

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

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

9.3.9 Concept Sentinel [iterators.sentinel]

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

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

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

(2.1) — i == s is well-defined.

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

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

9.3.10 Concept SizedSentinel [iterators.sizedsentinel]

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

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

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

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

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

3 [*Note:* disable_sized_sentinel provides a mechanism to enable use of sentinels and iterators with the library that meet the syntactic requirements but do not in fact satisfy SizedSentinel. A program that instantiates a library template that requires SizedSentinel with an iterator type I and sentinel type S that meet the syntactic requirements of SizedSentinel<S, I> but do not satisfy SizedSentinel is ill-formed with no diagnostic required unless disable_sized_sentinel<S, I> evaluates to true (6.2.1.3). — *end note*]

4 [*Note:* The SizedSentinel concept is satisfied by pairs of RandomAccessIterators (9.3.15) and by counted iterators and their sentinels (9.7.6.1). — *end note*]

9.3.11 Concept `InputIterator` [iterators.input]

1 The `InputIterator` concept is a refinement of `Iterator` (9.3.8). It defines requirements for a type whose referenced values can be read (from the requirement for `Readable` (9.3.4)) and which can be both pre- and post-incremented. [*Note:* Unlike in ISO/IEC 14882, input iterators are not required to satisfy `EqualityComparable` (7.4.3). — *end note*]

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

9.3.12 Concept `OutputIterator` [iterators.output]

1 The `OutputIterator` concept is a refinement of `Iterator` (9.3.8). It defines requirements for a type that can be used to write values (from the requirement for `Writable` (9.3.5)) and which can be both pre- and post-incremented. However, output iterators are not required to satisfy `EqualityComparable`.

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

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

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

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

9.3.13 Concept `ForwardIterator` [iterators.forward]

1 The `ForwardIterator` concept refines `InputIterator` (9.3.11), adding equality comparison and the multi-pass guarantee, specified below.

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

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

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

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

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

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

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

9.3.14 Concept `BidirectionalIterator` [iterators.bidirectional]

1 The `BidirectionalIterator` concept refines `ForwardIterator` (9.3.13), and adds the ability to move an iterator backward as well as forward.

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

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

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

(3.1) — `&--a == &a`.

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

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

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

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

9.3.15 Concept `RandomAccessIterator` [iterators.random.access]

1 The `RandomAccessIterator` concept refines `BidirectionalIterator` (9.3.14) and adds support for constant-time advancement with `+=`, `+`, `-=`, and `-`, and the computation of distance in constant time with `-`. Random access iterators also support array notation via subscripting.

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

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

(2.1) — `(a += n)` is equal to `b`.

(2.2) — `&(a += n)` is equal to `&a`.

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

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

(2.5) — `a + 0` is equal to `a`.

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

(2.7) — `(b += -n)` is equal to `a`.

(2.8) — `(b -= n)` is equal to `a`.

(2.9) — `&(b -= n)` is equal to `&b`.

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

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

9.4 Indirect callable requirements [indirectcallable]

9.4.1 In general [indirectcallable.general]

1 There are several concepts that group requirements of algorithms that take callable objects (ISO/IEC 14882:2014 §20.9.2) as arguments.

9.4.2 Indirect callables [indirectcallable.indirectinvocable]

1 The indirect callable concepts are used to constrain those algorithms that accept callable objects (ISO/IEC 14882:2014 §20.9.1) as arguments.

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

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

```
    result_of_t<F&(reference_t<I>&&)>>;

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

template <class F, class I1, class I2 = I1>
concept bool IndirectRelation =
  Readable<I1> && Readable<I2> &&
  CopyConstructible<F> &&
  Relation<F&, value_type_t<I1>&, value_type_t<I2>&> &&
  Relation<F&, value_type_t<I1>&, reference_t<I2>> &&
  Relation<F&, reference_t<I1>, value_type_t<I2>&> &&
  Relation<F&, reference_t<I1>, reference_t<I2>> &&
  Relation<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

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

template <class> struct indirect_result_of { };

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

9.4.3 Class template `projected` [projected]

1 The `projected` class template is intended for use when specifying the constraints of algorithms that accept callable objects and projections (3.3). It bundles a `Readable` type `I` and a function `Proj` into a new `Readable` type whose `reference` type is the result of applying `Proj` to the `reference_t` of `I`.

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

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

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

9.5 Common algorithm requirements [commonalgoreq]

9.5.1 In general [commonalgoreq.general]

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

2 [*Note:* The `equal_to<>` and `less<>` (8.3.2) function types used in the concepts below impose additional constraints on their arguments beyond those that appear explicitly in the concepts' bodies. `equal_to<>` requires its arguments satisfy `EqualityComparableWith` (7.4.3), and `less<>` requires its arguments satisfy `StrictTotallyOrderedWith` (7.4.4). — *end note*]

9.5.2 Concept `IndirectlyMovable` [commonalgoreq.indirectlymovable]

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

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

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

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

9.5.3 Concept `IndirectlyCopyable` [commonalgoreq.indirectlycopyable]

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

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

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

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

9.5.4 Concept `IndirectlySwappable` [commonalgoreq.indirectlyswappable]

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

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

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

9.5.5 Concept `IndirectlyComparable` [commonalgoreq.indirectlycomparable]

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

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

9.5.6 Concept `Permutable` [commonalgoreq.permutable]

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

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

9.5.7 Concept `Mergeable` [commonalgoreq.mergeable]

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

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

9.5.8 Concept `Sortable` [commonalgoreq.sortable]

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

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

9.6 Iterator primitives [iterator.primitives]

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

9.6.1 Iterator traits [iterator.traits]

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

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

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

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

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

```
    }
  }
```

— end example]

9.6.2 Standard iterator traits [iterator.stdtraits]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.6.3 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 can be used as compile time tags for algorithm selection. [*Note:* The preferred way to dispatch to more specialized algorithm implementations is with concept-based overloading. — *end note*] The category tags are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every input iterator of type `I`, `iterator_category_t<I>` shall be defined to be the most specific category tag that describes the iterator's behavior.

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

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

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

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

— *end example*]

9.6.4 Iterator operations [iterator.operations]

1 Since only types that satisfy `RandomAccessIterator` provide the `+` operator, and types that satisfy `SizedSentinel` provide the `-` operator, the library provides customization point objects (6.3.5.1) `advance`, `distance`, `next`, and `prev`. These customization point objects use `+` and `-` for random access iterators and ranges that satisfy `SizedSentinel` (and are, therefore, constant time for them); for output, input, forward and bidirectional iterators they use `++` to provide linear time implementations.

2 The name `advance` denotes a customization point object (6.3.5.1). It has the following function call operators:

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

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

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

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

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

6 *Effects:*

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

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

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

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

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

8 *Effects:*

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

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

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

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

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

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

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

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

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

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

Effects: Equivalent to: `return distance(ranges::begin(r), ranges::end(r));` (10.4)

Remarks: Instantiations of this member function template may be ill-formed if the declarations in `<experimental/ranges/range>` are not in scope at the point of instantiation (ISO/IEC 14882:2014 §14.6.4.1).

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

13 *Effects:* Equivalent to: `return ranges::size(r);` (10.5.1)

14 *Remarks:* Instantiations of this member function template may be ill-formed if the declarations in `<experimental/ranges/range>` are not in scope at the point of instantiation (ISO/IEC 14882:2014 §14.6.4.1).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.7 Iterator adaptors [iterators.predef]

9.7.1 Reverse iterators [iterators.reverse]

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

9.7.1.1 Class template `reverse_iterator` [reverse.iterator]

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

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

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

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

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

    friend constexpr rvalue_reference_t<I> iter_move(const reverse_iterator& i)
```

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

  private:
    I current; // exposition only
  };

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

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

9.7.1.2 reverse_iterator operations [reverse.iter.ops]

9.7.1.2.1 reverse_iterator constructor [reverse.iter.cons]

```
constexpr reverse_iterator();
```

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

```
explicit constexpr reverse_iterator(I x);
```

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

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

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

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

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

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

2 *Returns:* `*this`.

9.7.1.2.3 Conversion [reverse.iter.conv]

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

1 *Returns:* `current`.

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

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

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

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

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

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

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

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

1 *Effects:* `--current;`

2 *Returns:* `*this`.

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

3 *Effects:*

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

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

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

1 *Effects:* `++current`

2 *Returns:* `*this`.

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

3 *Effects:*

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

9.7.1.2.8 operator+ [reverse.iter.op+]

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

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

9.7.1.2.9 operator+= [reverse.iter.op+=]

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

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

2 *Returns:* `*this`.

9.7.1.2.10 operator- [reverse.iter.op-]

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

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

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

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

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

2 *Returns:* `*this`.

9.7.1.2.12 operator[] [reverse.iter.opindex]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.7.1.2.21 `iter_move` [reverse.iter.iter_move]

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

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

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

```
noexcept(ranges::iter_move(declval<I&>())) && noexcept(--declval<I&>()) &&
  is_nothrow_copy_constructible<I>::value
```

9.7.1.2.22 `iter_swap` [reverse.iter.iter_swap]

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

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

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

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

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

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

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

9.7.2 Insert iterators [iterators.insert]

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

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

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

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

9.7.2.1 Class template back_insert_iterator [back.insert.iterator]

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

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

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

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

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

9.7.2.2 back_insert_iterator operations [back.insert.iter.ops]

9.7.2.2.1 back_insert_iterator constructor [back.insert.iter.cons]

```
constexpr back_insert_iterator();
```

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

9.7.2.2.5 `back_inserter` [back.inserter]

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

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

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

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

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

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

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

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

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

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

```
constexpr front_insert_iterator();
```

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

9.7.2.4.5 `front_inserter` [front.inserter]

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

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

9.7.2.5 Class template `insert_iterator` [insert.iterator]

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

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

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

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

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

9.7.2.6 `insert_iterator` operations [insert.iter.ops]

9.7.2.6.1 `insert_iterator` constructor [insert.iter.cons]

```
insert_iterator();
```

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

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

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

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

9.7.2.6.2 `insert_iterator::operator=` [insert.iter.op=]

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

1 *Effects:* Equivalent to:

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

2 *Returns:* `*this`.

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

3 *Effects:* Equivalent to:

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

4 *Returns:* `*this`.

9.7.2.6.3 `insert_iterator::operator*` [insert.iter.op*]

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

1 *Returns:* `*this`.

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

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

1 *Returns:* `*this`.

9.7.2.6.5 `inserter` [inserter]

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

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

9.7.3 Move iterators and sentinels [iterators.move]

9.7.3.1 Class template `move_iterator` [move.iterator]

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

2 [*Example:*

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

— *end example*]

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

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

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

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

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

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

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

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

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

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

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

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

```
constexpr move_iterator();
```

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

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

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

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

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

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

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

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

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

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

1 *Returns:* `current`.

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

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

4 *Effects:* Equivalent to:

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

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

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

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

2 *Returns:* `*this`.

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

3 *Effects:* Equivalent to:

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

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

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

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

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

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

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

2 *Effects:* Equivalent to: `return !(x == y);`

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

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

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

4 *Effects:* Equivalent to: `return !(y < x);`

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

5 *Effects:* Equivalent to: `return y < x;`

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

6 *Effects:* Equivalent to: `return !(x < y);`.

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

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

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

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

2 *Effects:* Equivalent to: `return x + n;`

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

3 *Effects:* Equivalent to: `return ranges::iter_move(i.current);`

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

```
noexcept(ranges::iter_move(i.current))
```

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

5 *Effects:* Equivalent to: `ranges::iter_swap(x.current, y.current)`.

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

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

```
template <InputIterator I>
  constexpr move_iterator<I> make_move_iterator(I i);
```

7 *Returns:* `move_iterator<I>(i)`.

9.7.3.3 Class template `move_sentinel` [move.sentinel]

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

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

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

— *end example*]

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

    S base() const;

  private:
    S last; // exposition only
  };

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

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

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

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

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

```
constexpr move_sentinel();
```

1 *Effects:* Constructs a `move_sentinel`, value-initializing `last`. If `is_trivially_default_constructible<S>::value` is `true`, then this constructor is a `constexpr` constructor.

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

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

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

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

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

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

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

2 *Returns:* `*this`.

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

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

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

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

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

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

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

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

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

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

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

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

9.7.4 Common iterators [iterators.common]

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

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

3 [*Example:*

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

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

— *end example*]

9.7.4.1 Class template `common_iterator` [common.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <Iterator I, Sentinel<I> S>
    requires !Same<I, S>
  class common_iterator {
  public:
    using difference_type = difference_type_t<I>;

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

    decltype(auto) operator*();
    decltype(auto) operator*() const
      requires dereferenceable <const I>;
    decltype(auto) operator->() const
      requires see below;

    common_iterator& operator++();
    decltype(auto) operator++(int);
    common_iterator operator++(int)
      requires ForwardIterator<I>;

    friend rvalue_reference_t<I> iter_move(const common_iterator& i)
      noexcept(see below)
        requires InputIterator<I>;
    template <IndirectlySwappable<I> I2, class S2>
      friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
        noexcept(see below);
```

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

  template <Readable I, class S>
  struct value_type<common_iterator<I, S>> {
    using type = value_type_t<I>;
  };

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

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

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

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

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

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

`constexpr common_iterator();`

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

`constexpr common_iterator(I i);`

2 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `false`, `iter` with `i`, and value-initializing `sentinel`.

`constexpr common_iterator(S s);`

3 *Effects:* Constructs a `common_iterator`, initializing `is_sentinel` with `true`, value-initializing `iter`, and initializing `sentinel` with `s`.

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

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

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

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

1 *Effects:* Assigns `u.is_sentinel` to `is_sentinel`, `u.iter` to `iter`, and `u.sentinel` to `sentinel`.

2 *Returns:* `*this`

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

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

1 *Requires:* `!is_sentinel`

2 *Effects:* Equivalent to: `return *iter;`

9.7.4.2.4 `common_iterator::operator->` [common.iter.op.ref]

```
decltype(auto) operator->() const
  requires see below;
```

1 *Requires:* `!is_sentinel`

2 *Effects:* Equivalent to:

(2.1) — If `I` is a pointer type or if the expression `i.operator->()` is well-formed, `return iter;`

(2.2) — Otherwise, if the expression `*iter` is a glvalue:

```
auto&& tmp = *iter;
return addressof(tmp);
```

(2.3) — Otherwise, `return proxy(*iter);` where `proxy` is the exposition-only class:

```
class proxy {                 // exposition only
  value_type_t<I> keep_;
  proxy(reference_t<I>&& x)
    : keep_(std::move(x)) {}
public:
  const value_type_t<I>* operator->() const {
    return addressof(keep_);
  }
};
```

3 The expression in the requires clause is equivalent to:

```
Readable<const I> &&
  (requires(const I& i) { i.operator->(); } ||
   is_reference<reference_t<I>>::value ||
   Constructible<value_type_t<I>, reference_t<I>>)
```

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

```
common_iterator& operator++();
```

1 *Requires:* `!is_sentinel`

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

3 *Returns:* `*this`.

```
decltype(auto) operator++(int);
```

4 *Requires:* `!is_sentinel`.

5 *Effects:* Equivalent to: `return iter++;`

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

6 *Requires:* `!is_sentinel`

7 *Effects:* Equivalent to:

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

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

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

1 *Effects:* Equivalent to:

```
return x.is_sentinel ?
  (y.is_sentinel || y.iter == x.sentinel) :
  (!y.is_sentinel || x.iter == y.sentinel);
```

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

2 *Effects:* Equivalent to:

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

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

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

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

4 *Effects:* Equivalent to:

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

9.7.4.2.7 iter_move [common.iter.op.iter_move]

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

1 *Requires:* `!i.is_sentinel`.

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

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

```
noexcept(ranges::iter_move(i.iter))
```

9.7.4.2.8 iter_swap [common.iter.op.iter_swap]

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

1 *Requires:* `!x.is_sentinel && !y.is_sentinel`.

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

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

```
noexcept(ranges::iter_swap(x.iter, y.iter))
```

9.7.5 Default sentinels [default.sentinels]

9.7.5.1 Class default_sentinel [default.sent]

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

1 Class `default_sentinel` is an empty type used to denote the end of a range. It is intended to be used together with iterator types that know the bound of their range (e.g., `counted_iterator` (9.7.6.1)).

9.7.6 Counted iterators [iterators.counted]

9.7.6.1 Class template counted_iterator [counted.iterator]

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

2 [*Example:*

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

— *end example*]

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

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

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

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

    constexpr counted_iterator& operator++();
    decltype(auto) operator++(int);
    constexpr counted_iterator operator++(int)
      requires ForwardIterator<I>;
    constexpr counted_iterator& operator--()
      requires BidirectionalIterator<I>;
    constexpr counted_iterator operator--(int)
      requires BidirectionalIterator<I>;

    constexpr counted_iterator  operator+ (difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr counted_iterator& operator+=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr counted_iterator  operator- (difference_type n) const
      requires RandomAccessIterator<I>;
    constexpr counted_iterator& operator-=(difference_type n)
      requires RandomAccessIterator<I>;
    constexpr decltype(auto) operator[](difference_type n) const
      requires RandomAccessIterator<I>;

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

  private:
    I current; // exposition only
    difference_type_t<I> cnt; // exposition only
  };
```

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

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

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

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

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

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

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

9.7.6.2 `counted_iterator` operations [counted.iter.ops]

9.7.6.2.1 `counted_iterator` constructors [counted.iter.op.const]

```
constexpr counted_iterator();
```

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

```
constexpr counted_iterator(I i, difference_type_t<I> n);
```

2 *Requires:* `n >= 0`

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

```
constexpr counted_iterator(const counted_iterator<ConvertibleTo<I>>& i);
```

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

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

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

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

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

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

1 *Returns:* `current`.

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

```
constexpr difference_type_t<I> count() const;
```

1 *Returns:* `cnt`.

9.7.6.2.5 `counted_iterator::operator*` [counted.iter.op.star]

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

1 *Effects:* Equivalent to: `return *current;`

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

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

1 *Requires:* `cnt > 0`

2 *Effects:* Equivalent to:

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

3 *Returns:* `*this`.

```
decltype(auto) operator++(int);
```

4 *Requires:* `cnt > 0`.

5 *Effects:* Equivalent to:

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

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

6 *Requires:* `cnt > 0`

7 *Effects:* Equivalent to:

```
counted_iterator tmp = *this;
++*this;
return tmp;
```

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

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

1 *Effects:* Equivalent to:

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

2 *Returns:* `*this`.

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

3 *Effects:* Equivalent to:

```
counted_iterator tmp = *this;
--*this;
return tmp;
```

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

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

1 *Requires:* `n <= cnt`

2 *Effects:* Equivalent to: `return counted_iterator(current + n, cnt - n);`

9.7.6.2.9 `counted_iterator::operator+=` [counted.iter.op.+=]

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

1 *Requires:* `n <= cnt`

2 *Effects:*

```
current += n;
cnt -= n;
```

3 *Returns:* `*this`.

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

```
constexpr counted_iterator operator-(difference_type n) const
  requires RandomAccessIterator<I>;
```

1 *Requires:* `-n <= cnt`

2 *Effects:* Equivalent to: `return counted_iterator(current - n, cnt + n);`

9.7.6.2.11 `counted_iterator::operator-=` [counted.iter.op.-=]

```
constexpr counted_iterator& operator-=(difference_type n)
  requires RandomAccessIterator<I>;
```

1 *Requires:* `-n <= cnt`

2 *Effects:*

```
current -= n;
cnt += n;
```

3 *Returns:* `*this`.

9.7.6.2.12 `counted_iterator::operator[]` [counted.iter.op.index]

```
constexpr decltype(auto) operator[](difference_type n) const
  requires RandomAccessIterator<I>;
```

1 *Requires:* `n <= cnt`

2 *Effects:* Equivalent to: `return current[n];`

9.7.6.2.13 `counted_iterator` comparisons [counted.iter.op.comp]

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator==(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Requires:* `x` and `y` shall refer to elements of the same sequence (9.7.6).

2 *Effects:* Equivalent to: `return x.cnt == y.cnt;`

```
constexpr bool operator==(
  const counted_iterator<auto>& x, default_sentinel);
constexpr bool operator==(
  default_sentinel, const counted_iterator<auto>& x);
```

3 *Effects:* Equivalent to: `return x.cnt == 0;`

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator!=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
  constexpr bool operator!=(
    const counted_iterator<auto>& x, default_sentinel);
  constexpr bool operator!=(
    default_sentinel, const counted_iterator<auto>& x);
```

4 *Requires:* For the first overload, `x` and `y` shall refer to elements of the same sequence (9.7.6).

5 *Effects:* Equivalent to: `return !(x == y);`

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

6 *Requires:* `x` and `y` shall refer to elements of the same sequence (9.7.6).

7 *Effects:* Equivalent to: `return y.cnt < x.cnt;`

8 [*Note:* The argument order in the *Effects* element is reversed because `cnt` counts down, not up. — *end note*]

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator<=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

9 *Requires:* `x` and `y` shall refer to elements of the same sequence (9.7.6).

10 *Effects:* Equivalent to: `return !(y < x);`

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

11 *Requires:* `x` and `y` shall refer to elements of the same sequence (9.7.6).

12 *Effects:* Equivalent to: `return y < x;`

```
template <class I1, class I2>
    requires Common<I1, I2>
  constexpr bool operator>=(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

13 *Requires:* `x` and `y` shall refer to elements of the same sequence (9.7.6).

14 *Effects:* Equivalent to: `return !(x < y);`

9.7.6.2.14 `counted_iterator` non-member functions [counted.iter.nonmember]

```
  template <class I1, class I2>
      requires Common<I1, I2>
  constexpr difference_type_t<I2> operator-(
    const counted_iterator<I1>& x, const counted_iterator<I2>& y);
```

1 *Requires:* `x` and `y` shall refer to elements of the same sequence (9.7.6).

2 *Effects:* Equivalent to: `return y.cnt - x.cnt;`

```
template <class I>
  constexpr difference_type_t<I> operator-(
    const counted_iterator<I>& x, default_sentinel y);
```

3 *Effects:* Equivalent to: `return -x.cnt;`

```
template <class I>
  constexpr difference_type_t<I> operator-(
    default_sentinel x, const counted_iterator<I>& y);
```

4 *Effects:* Equivalent to: `return y.cnt;`

```
template <RandomAccessIterator I>
  constexpr counted_iterator<I> operator+(
    difference_type_t<I> n, const counted_iterator<I>& x);
```

5 *Requires:* `n <= x.cnt`.

6 *Effects:* Equivalent to: `return x + n;`

```
friend constexpr rvalue_reference_t<I> iter_move(const counted_iterator& i)
  noexcept(see below)
    requires InputIterator<I>;
```

7 *Effects:* Equivalent to: `return ranges::iter_move(i.current);`

8 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_move(i.current))
```

```
template <IndirectlySwappable<I> I2>
  friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
    noexcept(see below);
```

9 *Effects:* Equivalent to `ranges::iter_swap(x.current, y.current)`.

10 *Remarks:* The expression in `noexcept` is equivalent to:

```
noexcept(ranges::iter_swap(x.current, y.current))
```

```
template <Iterator I>
  constexpr counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);
```

11 *Requires:* `n >= 0`.

12 *Returns:* `counted_iterator<I>(i, n)`.

9.7.7 Dangling wrapper [dangling.wrappers]

9.7.7.1 Class template `dangling` [dangling.wrap]

1 Class template `dangling` is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <CopyConstructible T>
  class dangling {
  public:
    constexpr dangling() requires DefaultConstructible<T>;
    constexpr dangling(T t);
    constexpr T get_unsafe() const;
  private:
    T value; // exposition only
  };

  template <Range R>
  using safe_iterator_t =
    conditional_t<is_lvalue_reference<R>::value,
      iterator_t<R>,
      dangling<iterator_t<R>>>;
}}}}
```

9.7.7.2 `dangling` operations [dangling.wrap.ops]

9.7.7.2.1 `dangling` constructors [dangling.wrap.op.const]

```
constexpr dangling() requires DefaultConstructible<T>;
```

1 *Effects:* Constructs a `dangling`, value-initializing `value`.

```
constexpr dangling(T t);
```

2 *Effects:* Constructs a `dangling`, initializing `value` with `t`.

9.7.7.2.2 `dangling::get_unsafe` [dangling.wrap.op.get]

```
constexpr T get_unsafe() const;
```

1 *Returns:* `value`.

9.7.8 Unreachable sentinel [unreachable.sentinels]

9.7.8.1 Class `unreachable` [unreachable.sentinel]

1 Class `unreachable` is a sentinel type that can be used with any `Iterator` to denote an infinite range. Comparing an iterator for equality with an object of type `unreachable` always returns `false`.

[*Example:*

```
char* p;
// set p to point to a character buffer containing newlines
char* nl = find(p, unreachable(), '\n');
```

Provided a newline character really exists in the buffer, the use of `unreachable` above potentially makes the call to `find` more efficient since the loop test against the sentinel does not require a conditional branch. — *end example*]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  class unreachable { };

  template <Iterator I>
    constexpr bool operator==(const I&, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator==(unreachable, const I&) noexcept;
  template <Iterator I>
    constexpr bool operator!=(const I&, unreachable) noexcept;
  template <Iterator I>
    constexpr bool operator!=(unreachable, const I&) noexcept;
}}}}
```

9.7.8.2 `unreachable` operations [unreachable.sentinel.ops]

9.7.8.2.1 `operator==` [unreachable.sentinel.op==]

```
template <Iterator I>
  constexpr bool operator==(const I&, unreachable) noexcept;
template <Iterator I>
  constexpr bool operator==(unreachable, const I&) noexcept;
```

1 *Returns:* `false`.

9.7.8.2.2 operator!= [unreachable.sentinel.op!=]

```
template <Iterator I>
  constexpr bool operator!=(const I& x, unreachable y) noexcept;
template <Iterator I>
  constexpr bool operator!=(unreachable x, const I& y) noexcept;
```

1 *Returns:* `true`.

9.8 Stream iterators [iterators.stream]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[*Example:*

```
partial_sum(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. — *end example*]

9.8.1 Class template istream_iterator [istream.iterator]

1 The class template `istream_iterator` is an input iterator (9.3.11) that reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the iterator fails to read and store a value of `T` (`fail()` on the stream returns `true`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-stream iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

2 Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class T, class charT = char, class traits = char_traits<charT>,
      class Distance = ptrdiff_t>
  class istream_iterator {
  public:
    typedef input_iterator_tag iterator_category;
    typedef Distance difference_type;
    typedef T value_type;
    typedef const T& reference;
    typedef const T* pointer;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT, traits> istream_type;
    constexpr istream_iterator();
    constexpr istream_iterator(default_sentinel);
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator& x) = default;
    ~istream_iterator() = default;
```

```
    const T& operator*() const;
    const T* operator->() const;
    istream_iterator& operator++();
    istream_iterator  operator++(int);
  private:
    basic_istream<charT, traits>* in_stream; // exposition only
    T value;                                 // exposition only
  };

  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator==(default_sentinel x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T, charT, traits, Distance>& x,
            default_sentinel y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(default_sentinel x,
            const istream_iterator<T, charT, traits, Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
            default_sentinel y);
}}}}
```

9.8.1.1 `istream_iterator` constructors and destructor [istream.iterator.cons]

```
constexpr istream_iterator();
constexpr istream_iterator(default_sentinel);
```

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then these constructors shall be `constexpr` constructors.

2 *Postcondition:* `in_stream == nullptr`.

```
istream_iterator(istream_type& s);
```

3 *Effects:* Initializes `in_stream` with `&s`. `value` may be initialized during construction or the first time it is referenced.

4 *Postcondition:* `in_stream == &s`.

```
istream_iterator(const istream_iterator& x) = default;
```

5 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6 *Postcondition:* `in_stream == x.in_stream`.

```
~istream_iterator() = default;
```

7 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

9.8.1.2 istream_iterator operations [istream.iterator.ops]

```
const T& operator*() const;
```

1 *Returns:* `value`.

```
const T* operator->() const;
```

2 *Effects:* Equivalent to: `return addressof(operator*())`.

```
istream_iterator& operator++();
```

3 *Requires:* `in_stream != nullptr`.

4 *Effects:* `*in_stream >> value`.

5 *Returns:* `*this`.

```
istream_iterator operator++(int);
```

6 *Requires:* `in_stream != nullptr`.

7 *Effects:*

```
istream_iterator tmp = *this;
*in_stream >> value;
return tmp;
```

```
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance> &x,
                  const istream_iterator<T, charT, traits, Distance> &y);
```

8 *Returns:* `x.in_stream == y.in_stream`.

```
template <class T, class charT, class traits, class Distance>
  bool operator==(default_sentinel x,
                  const istream_iterator<T, charT, traits, Distance> &y);
```

9 *Returns:* `nullptr == y.in_stream`.

```
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T, charT, traits, Distance> &x,
                  default_sentinel y);
```

10 *Returns:* `x.in_stream == nullptr`.

```
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
                  const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(default_sentinel x,
                  const istream_iterator<T, charT, traits, Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T, charT, traits, Distance>& x,
                  default_sentinel y);
```

11 *Returns:* `!(x == y)`

9.8.2 Class template ostream_iterator [ostream.iterator]

1 `ostream_iterator` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. If it was constructed with `charT*` as a constructor argument, this string, called a *delimiter string*, is written to the stream after every `T` is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

2 `ostream_iterator` is defined as:

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class T, class charT = char, class traits = char_traits<charT>>
  class ostream_iterator {
  public:
    typedef ptrdiff_t difference_type;
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT, traits> ostream_type;
    constexpr ostream_iterator() noexcept;
    ostream_iterator(ostream_type& s) noexcept;
    ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;
    ostream_iterator(const ostream_iterator& x) noexcept;
    ~ostream_iterator();
    ostream_iterator& operator=(const T& value);

    ostream_iterator& operator*();
    ostream_iterator& operator++();
    ostream_iterator& operator++(int);
  private:
    basic_ostream<charT, traits>* out_stream;  // exposition only
    const charT* delim;                         // exposition only
  };
}}}}
```

9.8.2.1 ostream_iterator constructors and destructor [ostream.iterator.cons.des]

`constexpr ostream_iterator() noexcept;`

1 *Effects:* Initializes `out_stream` and `delim` with `nullptr`.

`ostream_iterator(ostream_type& s) noexcept;`

2 *Effects:* Initializes `out_stream` with `&s` and `delim` with `nullptr`.

`ostream_iterator(ostream_type& s, const charT* delimiter) noexcept;`

3 *Effects:* Initializes `out_stream` with `&s` and `delim` with `delimiter`.

`ostream_iterator(const ostream_iterator& x) noexcept;`

4 *Effects:* Constructs a copy of `x`.

`~ostream_iterator();`

5 *Effects:* The iterator is destroyed.

9.8.2.2 `ostream_iterator` operations [ostream.iterator.ops]

```
ostream_iterator& operator=(const T& value);
```

1 *Effects:* Equivalent to:

```
*out_stream << value;
if(delim != nullptr)
  *out_stream << delim;
return *this;
```

```
ostream_iterator& operator*();
```

2 *Returns:* `*this`.

```
ostream_iterator& operator++();
ostream_iterator& operator++(int);
```

3 *Returns:* `*this`.

9.8.3 Class template `istreambuf_iterator` [istreambuf.iterator]

1 The class template `istreambuf_iterator` defines an input iterator (9.3.11) that reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (`streambuf_type::sgetc()` returns `traits::eof()`), the iterator becomes equal to the *end-of-stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(nullptr)` both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class charT, class traits = char_traits<charT>>
  class istreambuf_iterator {
  public:
    typedef input_iterator_tag             iterator_category;
    typedef charT                          value_type;
    typedef typename traits::off_type      difference_type;
    typedef charT                          reference;
    typedef unspecified                    pointer;
    typedef charT                          char_type;
    typedef traits                         traits_type;
    typedef typename traits::int_type      int_type;
    typedef basic_streambuf<charT, traits> streambuf_type;
    typedef basic_istream<charT, traits>   istream_type;

    class proxy;                           // exposition only

    constexpr istreambuf_iterator() noexcept;
    constexpr istreambuf_iterator(default_sentinel) noexcept;
    istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
    ~istreambuf_iterator() = default;
    istreambuf_iterator(istream_type& s) noexcept;
    istreambuf_iterator(streambuf_type* s) noexcept;
    istreambuf_iterator(const proxy& p) noexcept;
```

```
    charT operator*() const;
    istreambuf_iterator& operator++();
    proxy operator++(int);
    bool equal(const istreambuf_iterator& b) const;
  private:
    streambuf_type* sbuf_;                // exposition only
  };

  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT, traits>& a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator==(default_sentinel a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT, traits>& a,
            default_sentinel b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT, traits>& a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator!=(default_sentinel a,
            const istreambuf_iterator<charT, traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT, traits>& a,
            default_sentinel b);
}}}}
```

9.8.3.1 Class template `istreambuf_iterator::proxy` [istreambuf.iterator::proxy]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class charT, class traits = char_traits<charT>>
  class istreambuf_iterator<charT, traits>::proxy { // exposition only
    charT keep_;
    basic_streambuf<charT, traits>* sbuf_;
    proxy(charT c, basic_streambuf<charT, traits>* sbuf)
      : keep_(c), sbuf_(sbuf) { }
  public:
    charT operator*() { return keep_; }
  };
}}}}
```

1 Class `istreambuf_iterator<charT, traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

9.8.3.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

```
constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(default_sentinel) noexcept;
```

1 *Effects:* Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT, traits>& s) noexcept;
istreambuf_iterator(basic_streambuf<charT, traits>* s) noexcept;
```

2 *Effects:* Constructs an `istreambuf_iterator` that uses the `basic_streambuf` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

```
istreambuf_iterator(const proxy& p) noexcept;
```

3 *Effects:* Constructs a `istreambuf_iterator` that uses the `basic_streambuf` object pointed to by the `proxy` object's constructor argument `p`.

9.8.3.3 `istreambuf_iterator::operator*` [istreambuf.iterator::op*]

```
charT operator*() const
```

1 *Returns:* The character obtained via the `streambuf` member `sbuf_->sgetc()`.

9.8.3.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```
istreambuf_iterator&
    istreambuf_iterator<charT, traits>::operator++();
```

1 *Effects:* Equivalent to `sbuf_->sbumpc()`.

2 *Returns:* `*this`.

```
proxy istreambuf_iterator<charT, traits>::operator++(int);
```

3 *Effects:* Equivalent to: `return proxy(sbuf_->sbumpc(), sbuf_);`

9.8.3.5 `istreambuf_iterator::equal` [istreambuf.iterator::equal]

```
bool equal(const istreambuf_iterator& b) const;
```

1 *Returns:* `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

9.8.3.6 `operator==` [istreambuf.iterator::op==]

```
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT, traits>& a,
                  const istreambuf_iterator<charT, traits>& b);
```

1 *Effects:* Equivalent to: `return a.equal(b);`

```
template <class charT, class traits>
  bool operator==(default_sentinel a,
                  const istreambuf_iterator<charT, traits>& b);
```

2 *Effects:* Equivalent to: `return istreambuf_iterator<charT, traits>{}.equal(b);`

```
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT, traits>& a,
                  default_sentinel b);
```

3 *Effects:* Equivalent to: `return a.equal(istreambuf_iterator<charT, traits>{});`

9.8.3.7 `operator!=` [istreambuf.iterator::op!=]

```
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT, traits>& a,
                  const istreambuf_iterator<charT, traits>& b);
template <class charT, class traits>
  bool operator!=(default_sentinel a,
                  const istreambuf_iterator<charT, traits>& b);
```

```
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT, traits>& a,
                  default_sentinel b);
```

1 *Effects:* Equivalent to: `return !(a == b);`

9.8.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class charT, class traits = char_traits<charT>>
  class ostreambuf_iterator {
  public:
    typedef ptrdiff_t                     difference_type;
    typedef charT                         char_type;
    typedef traits                        traits_type;
    typedef basic_streambuf<charT, traits> streambuf_type;
    typedef basic_ostream<charT, traits>   ostream_type;

    constexpr ostreambuf_iterator() noexcept;
    ostreambuf_iterator(ostream_type& s) noexcept;
    ostreambuf_iterator(streambuf_type* s) noexcept;
    ostreambuf_iterator& operator=(charT c);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);
    bool failed() const noexcept;

  private:
    streambuf_type* sbuf_;                // exposition only
  };
}}}}
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

9.8.4.1 `ostreambuf_iterator` constructors [ostreambuf.iter.cons]

```
constexpr ostreambuf_iterator() noexcept;
```

1 *Effects:* Initializes `sbuf_` with `nullptr`.

```
ostreambuf_iterator(ostream_type& s) noexcept;
```

2 *Requires:* `s.rdbuf() != nullptr`.

3 *Effects:* Initializes `sbuf_` with `s.rdbuf()`.

```
ostreambuf_iterator(streambuf_type* s) noexcept;
```

4 *Requires:* `s != nullptr`.

5 *Effects:* Initializes `sbuf_` with `s`.

9.8.4.2 `ostreambuf_iterator` operations [ostreambuf.iter.ops]

```
ostreambuf_iterator&
  operator=(charT c);
```

1 *Requires:* `sbuf_ != nullptr`.

2 *Effects:* If `failed()` yields `false`, calls `sbuf_->sputc(c)`; otherwise has no effect.

3 *Returns:* `*this`.

```
ostreambuf_iterator& operator*();
```

4 *Returns:* `*this`.

```
ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);
```

5 *Returns:* `*this`.

```
bool failed() const noexcept;
```

6 *Requires:* `sbuf_ != nullptr`.

7 *Returns:* `true` if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

10 Ranges library [ranges]

10.1 General [ranges.general]

1 This Clause describes components for dealing with ranges of elements.

2 The following subclauses describe range and view requirements, and components for range primitives as summarized in Table 9.

Table 9 — Ranges library summary

	Subclause	Header(s)
10.4	Range access	<experimental/ranges/range>
10.5	Range primitives	
10.6	Requirements	

10.2 decay_copy [ranges.decaycopy]

1 Several places in this Clause use the expression *DECAY_COPY*(x), which is expression-equivalent to:

```
decay_t<decltype((x))>(x)
```

10.3 Header <experimental/ranges/range> synopsis [range.synopsis]

```
#include <experimental/ranges/iterator>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 10.4, range access:
  namespace {
    constexpr unspecified  begin = unspecified ;
    constexpr unspecified  end = unspecified ;
    constexpr unspecified  cbegin = unspecified ;
    constexpr unspecified  cend = unspecified ;
    constexpr unspecified  rbegin = unspecified ;
    constexpr unspecified  rend = unspecified ;
    constexpr unspecified  crbegin = unspecified ;
    constexpr unspecified  crend = unspecified ;
  }

  // 10.5, range primitives:
  namespace {
    constexpr unspecified  size = unspecified ;
    constexpr unspecified  empty = unspecified ;
    constexpr unspecified  data = unspecified ;
    constexpr unspecified  cdata = unspecified ;
  }

  template <class T>
  using iterator_t = decltype(ranges::begin(declval<T&>()));

  template <class T>
  using sentinel_t = decltype(ranges::end(declval<T&>()));
```

```
template <class>
constexpr bool disable_sized_range = false;

template <class T>
struct enable_view { };

struct view_base { };

// 10.6, range requirements:

// 10.6.2, Range:
template <class T>
concept bool Range = see below;

// 10.6.3, SizedRange:
template <class T>
concept bool SizedRange = see below;

// 10.6.4, View:
template <class T>
concept bool View = see below;

// 10.6.5, BoundedRange:
template <class T>
concept bool BoundedRange = see below;

// 10.6.6, InputRange:
template <class T>
concept bool InputRange = see below;

// 10.6.7, OutputRange:
template <class R, class T>
concept bool OutputRange = see below;

// 10.6.8, ForwardRange:
template <class T>
concept bool ForwardRange = see below;

// 10.6.9, BidirectionalRange:
template <class T>
concept bool BidirectionalRange = see below;

// 10.6.10, RandomAccessRange:
template <class T>
concept bool RandomAccessRange = see below;
}}}}
```

10.4 Range access [range.access]

1 In addition to being available via inclusion of the `<experimental/ranges/range>` header, the customization point objects in 10.4 are available when `<experimental/ranges/iterator>` is included.

10.4.1 `begin` [range.access.begin]

1 The name `begin` denotes a customization point object (6.3.5.1). The expression `ranges::begin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::begin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::begin(E)` behaves similarly to `std::begin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, `(E) + 0` if `E` has array type (ISO/IEC 14882:2014 §3.9.2).

(1.3) — Otherwise, *`DECAY_COPY`*`((E).begin())` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, *`DECAY_COPY`*`(begin(E))` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>` with overload resolution performed in a context that includes the declaration `void begin(auto&) = delete;` and does not include a declaration of `ranges::begin`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, `ranges::begin(E)` is ill-formed.

2 [*Note:* Whenever `ranges::begin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

10.4.2 `end` [range.access.end]

1 The name `end` denotes a customization point object (6.3.5.1). The expression `ranges::end(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::end(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::end(E)` behaves similarly to `std::end(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, `(E) + extent<T>::value` if `E` has array type (ISO/IEC 14882:2014 §3.9.2) `T`.

(1.3) — Otherwise, *`DECAY_COPY`*`((E).end())` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(ranges::begin(E))>`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, *`DECAY_COPY`*`(end(E))` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(ranges::begin(E))>` with overload resolution performed in a context that includes the declaration `void end(auto&) = delete;` and does not include a declaration of `ranges::end`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, `ranges::end(E)` is ill-formed.

2 [*Note:* Whenever `ranges::end(E)` is a valid expression, the types of `ranges::end(E)` and `ranges::begin(E)` satisfy `Sentinel`. — *end note*]

10.4.3 `cbegin` [range.access.cbegin]

1 The name `cbegin` denotes a customization point object (6.3.5.1). The expression `ranges::cbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to `ranges::begin(static_cast<const T&>(E))`.

2 Use of `ranges::cbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::cbegin(E)` behaves similarly to `std::cbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::cbegin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

10.4.4 cend [range.access.cend]

1 The name cend denotes a customization point object (6.3.5.1). The expression ranges::cend(E) for some subexpression E of type T is expression-equivalent to ranges::end(static_cast<const T&>(E)).

2 Use of ranges::cend(E) with rvalue E is deprecated. [*Note:* This deprecated usage exists so that ranges::cend(E) behaves similarly to std::cend(E) as defined in ISO/IEC 14882 when E is an rvalue. — *end note*]

3 [*Note:* Whenever ranges::cend(E) is a valid expression, the types of ranges::cend(E) and ranges::cbegin(E) satisfy Sentinel. — *end note*]

10.4.5 rbegin [range.access.rbegin]

1 The name rbegin denotes a customization point object (6.3.5.1). The expression ranges::rbegin(E) for some subexpression E is expression-equivalent to:

(1.1) — ranges::rbegin(static_cast<const T&>(E)) if E is an rvalue of type T. This usage is deprecated. [*Note:* This deprecated usage exists so that ranges::rbegin(E) behaves similarly to std::rbegin(E) as defined in ISO/IEC 14882 when E is an rvalue. — *end note*]

(1.2) — Otherwise, *DECAY_COPY*((E).rbegin()) if it is a valid expression and its type I meets the syntactic requirements of Iterator<I>. If Iterator is not satisfied, the program is ill-formed with no diagnostic required.

(1.3) — Otherwise, make_reverse_iterator(ranges::end(E)) if both ranges::begin(E) and ranges::end(E) are valid expressions of the same type I which meets the syntactic requirements of BidirectionalIterator<I> (9.3.14).

(1.4) — Otherwise, ranges::rbegin(E) is ill-formed.

2 [*Note:* Whenever ranges::rbegin(E) is a valid expression, its type satisfies Iterator. — *end note*]

10.4.6 rend [range.access.rend]

1 The name rend denotes a customization point object (6.3.5.1). The expression ranges::rend(E) for some subexpression E is expression-equivalent to:

(1.1) — ranges::rend(static_cast<const T&>(E)) if E is an rvalue of type T. This usage is deprecated. [*Note:* This deprecated usage exists so that ranges::rend(E) behaves similarly to std::rend(E) as defined in ISO/IEC 14882 when E is an rvalue. — *end note*]

(1.2) — Otherwise, *DECAY_COPY*((E).rend()) if it is a valid expression and its type S meets the syntactic requirements of Sentinel<S, decltype(ranges::rbegin(E))>. If Sentinel is not satisfied, the program is ill-formed with no diagnostic required.

(1.3) — Otherwise, make_reverse_iterator(ranges::begin(E)) if both ranges::begin(E) and ranges::end(E) are valid expressions of the same type I which meets the syntactic requirements of BidirectionalIterator<I> (9.3.14).

(1.4) — Otherwise, ranges::rend(E) is ill-formed.

2 [*Note:* Whenever ranges::rend(E) is a valid expression, the types of ranges::rend(E) and ranges::rbegin(E) satisfy Sentinel. — *end note*]

10.4.7 `crbegin` [range.access.crbegin]

1 The name `crbegin` denotes a customization point object (6.3.5.1). The expression `ranges::crbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to `ranges::rbegin(static_cast<const T&>(E))`.

2 Use of `ranges::crbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::crbegin(E)` behaves similarly to `std::crbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::crbegin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

10.4.8 `crend` [range.access.crend]

1 The name `crend` denotes a customization point object (6.3.5.1). The expression `ranges::crend(E)` for some subexpression `E` of type `T` is expression-equivalent to `ranges::rend(static_cast<const T&>(E))`.

2 Use of `ranges::crend(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::crend(E)` behaves similarly to `std::crend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::crend(E)` is a valid expression, the types of `ranges::crend(E)` and `ranges::crbegin(E)` satisfy `Sentinel`. — *end note*]

10.5 Range primitives [range.primitives]

1 In addition to being available via inclusion of the `<experimental/ranges/range>` header, the customization point objects in 10.5 are available when `<experimental/ranges/iterator>` is included.

10.5.1 `size` [range.primitives.size]

1 The name `size` denotes a customization point object (6.3.5.1). The expression `ranges::size(E)` for some subexpression `E` with type `T` is expression-equivalent to:

(1.1) — *`DECAY_COPY`*`(extent<T>::value)` if `T` is an array type (ISO/IEC 14882:2014 §3.9.2).

(1.2) — Otherwise, *`DECAY_COPY`*`(static_cast<const T&>(E).size())` if it is a valid expression and its type `I` satisfies `Integral<I>` and `disable_sized_range<T>` (10.6.3) is `false`.

(1.3) — Otherwise, *`DECAY_COPY`*`(size(static_cast<const T&>(E)))` if it is a valid expression and its type `I` satisfies `Integral<I>` with overload resolution performed in a context that includes the declaration `void size(const auto&) = delete;` and does not include a declaration of `ranges::size`, and `disable_sized_range<T>` is `false`.

(1.4) — Otherwise, *`DECAY_COPY`*`(ranges::cend(E) - ranges::cbegin(E))`, except that `E` is only evaluated once, if it is a valid expression and the types `I` and `S` of `ranges::cbegin(E)` and `ranges::cend(E)` meet the syntactic requirements of `SizedSentinel<S, I>` (9.3.10) and `ForwardIterator<I>`. If `SizedSentinel` and `ForwardIterator` are not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, `ranges::size(E)` is ill-formed.

2 [*Note:* Whenever `ranges::size(E)` is a valid expression, its type satisfies `Integral`. — *end note*]

10.5.2 `empty` [range.primitives.empty]

1 The name `empty` denotes a customization point object (6.3.5.1). The expression `ranges::empty(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `bool((E).empty())` if it is a valid expression.

(1.2) — Otherwise, `ranges::size(E) == 0` if it is a valid expression.

(1.3) — Otherwise, `bool(ranges::begin(E) == ranges::end(E))`, except that `E` is only evaluated once, if it is a valid expression and the type of `ranges::begin(E)` satisfies `ForwardIterator`.

(1.4) — Otherwise, `ranges::empty(E)` is ill-formed.

2 [*Note:* Whenever `ranges::empty(E)` is a valid expression, it has type `bool`. — *end note*]

10.5.3 `data` [range.primitives.data]

1 The name `data` denotes a customization point object (6.3.5.1). The expression `ranges::data(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `ranges::data(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::data(E)` behaves similarly to `std::data(E)` as defined in the C++ Working Paper when `E` is an rvalue. — *end note*]

(1.2) — Otherwise, *`DECAY_COPY`*`((E).data())` if it is a valid expression of pointer to object type.

(1.3) — Otherwise, `ranges::begin(E)` if it is a valid expression of pointer to object type.

(1.4) — Otherwise, `ranges::data(E)` is ill-formed.

2 [*Note:* Whenever `ranges::data(E)` is a valid expression, it has pointer to object type. — *end note*]

10.5.4 `cdata` [range.primitives.cdata]

1 The name `cdata` denotes a customization point object (6.3.5.1). The expression `ranges::cdata(E)` for some subexpression `E` of type `T` is expression-equivalent to `ranges::data(static_cast<const T&>(E))`.

2 Use of `ranges::cdata(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::cdata(E)` has behavior consistent with `ranges::data(E)` when `E` is an rvalue. — *end note*]

3 [*Note:* Whenever `ranges::cdata(E)` is a valid expression, it has pointer to object type. — *end note*]

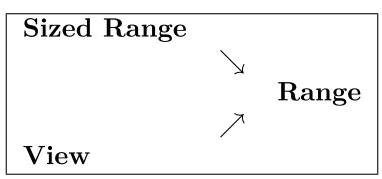
10.6 Range requirements [ranges.requirements]

10.6.1 In general [ranges.requirements.general]

1 Ranges are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. It their simplest form, a range object is one on which one can call `begin` and `end` to get an iterator (9.3.8) and a sentinel (9.3.9). To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of ranges.

2 This document defines three fundamental categories of ranges based on the syntax and semantics supported by each: *range*, *sized range* and *view*, as shown in Table 10.

Table 10 — Relations among range categories



3 The `Range` concept requires only that `begin` and `end` return an iterator and a sentinel. The `SizedRange` concept refines `Range` with the requirement that the number of elements in the range can be determined in constant time using the `size` function. The `View` concept specifies requirements on a `Range` type with constant-time copy and assign operations.

4 In addition to the three fundamental range categories, this document defines a number of convenience refinements of `Range` that group together requirements that appear often in the concepts and algorithms. *Bounded ranges* are ranges for which `begin` and `end` return objects of the same type. *Random access ranges* are ranges for which `begin` returns a type that satisfies `RandomAccessIterator` (9.3.15). The range categories *bidirectional ranges*, *forward ranges*, *input ranges*, and *output ranges* are defined similarly.

10.6.2 Ranges [ranges.range]

1 The `Range` concept defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and an `end` sentinel. [*Note:* Most algorithms requiring this concept simply forward to an `Iterator`-based algorithm by calling `begin` and `end`. — *end note*]

```
template <class T>
concept bool Range =
  requires(T&& t) {
    ranges::begin(t); // not necessarily equality-preserving (see below)
    ranges::end(t);
  };
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `Range<T>` is satisfied only if

(2.1) — `[begin(t),end(t))` denotes a range.

(2.2) — Both `begin(t)` and `end(t)` are amortized constant time and non-modifying. [*Note:* `begin(t)` and `end(t)` do not require implicit expression variations (7.1.1). — *end note*]

(2.3) — If `iterator_t<T>` satisfies `ForwardIterator`, `begin(t)` is equality preserving.

3 [*Note:* Equality preservation of both `begin` and `end` enables passing a `Range` whose iterator type satisfies `ForwardIterator` to multiple algorithms and making multiple passes over the range by repeated calls to `begin` and `end`. Since `begin` is not required to be equality preserving when the return type does not satisfy `ForwardIterator`, repeated calls might not return equal values or might not be well-defined; `begin` should be called at most once for such a range. — *end note*]

10.6.3 Sized ranges [ranges.sized]

1 The `SizedRange` concept specifies the requirements of a `Range` type that knows its size in constant time with the `size` function.

```
template <class T>
concept bool SizedRange =
  Range<T> &&
  !disable_sized_range<remove_cv_t<remove_reference_t<T>>> &&
  requires(T& t) {
    { ranges::size(t) } -> ConvertibleTo<difference_type_t<iterator_t<T>>>;
  };
```

2 Given an lvalue `t` of type `remove_reference_t<T>`, `SizedRange<T>` is satisfied only if:

(2.1) — `ranges::size(t)` is $\mathcal{O}(1)$, does not modify `t`, and is equal to `ranges::distance(t)`.

(2.2) — If `iterator_t<T>` satisfies `ForwardIterator`, `size(t)` is well-defined regardless of the evaluation of `begin(t)`. [*Note:* `size(t)` is otherwise not required be well-defined after evaluating `begin(t)`. For a `SizedRange` whose iterator type does not model `ForwardIterator`, for example, `size(t)` might only be well-defined if evaluated before the first call to `begin(t)`. — *end note*]

3 [*Note:* The `disable_sized_range` predicate provides a mechanism to enable use of range types with the library that meet the syntactic requirements but do not in fact satisfy `SizedRange`. A program that instantiates a library template that requires a `Range` with such a range type `R` is ill-formed with no diagnostic required unless `disable_sized_range<remove_cv_t<remove_reference_t<R>>>` evaluates to `true` (6.2.1.3). — *end note*]

10.6.4 Views [ranges.view]

1 The `View` concept specifies the requirements of a `Range` type that has constant time copy, move and assignment operators; that is, the cost of these operations is not proportional to the number of elements in the `View`.

2 [*Example:* Examples of `View`s are:

(2.1) — A `Range` type that wraps a pair of iterators.

(2.2) — A `Range` type that holds its elements by `shared_ptr` and shares ownership with all its copies.

(2.3) — A `Range` type that generates its elements on demand.

A container (ISO/IEC 14882:2014 §23) is not a `View` since copying the container copies the elements, which cannot be done in constant time. — *end example*]

```
template <class T>
constexpr bool view-predicate   // exposition only
  = see below;

template <class T>
concept bool View =
  Range<T> &&
  Semiregular<T> &&
  view-predicate <T>;
```

3 Since the difference between `Range` and `View` is largely semantic, the two are differentiated with the help of the `enable_view` trait. Users may specialize `enable_view` to derive from `true_type` or `false_type`.

4 For a type `T`, the value of ***`view-predicate`*** `<T>` shall be:

(4.1) — If `enable_view<T>` has a member type `type`, `enable_view<T>::type::value`;

(4.2) — Otherwise, if `T` is derived from `view_base`, `true`;

(4.3) — Otherwise, if `T` is an instantiation of class template `initializer_list` (ISO/IEC 14882:2014 § 18.9), `set` (ISO/IEC 14882:2014 §23.4.6), `multiset` (ISO/IEC 14882:2014 §23.4.7), `unordered_set` (ISO/IEC 14882:2014 §23.5.6), or `unordered_multiset` (ISO/IEC 14882:2014 §23.5.7), `false`;

(4.4) — Otherwise, if both `T` and `const T` satisfy `Range` and `reference_t <iterator_t<T>>` is not the same type as `reference_t <iterator_t<const T>>`, `false`; [*Note:* Deep `const`-ness implies element ownership, whereas shallow `const`-ness implies reference semantics. — *end note*]

(4.5) — Otherwise, `true`.

10.6.5 Bounded ranges [ranges.bounded]

1 The `BoundedRange` concept specifies requirements of a `Range` type for which `begin` and `end` return objects of the same type. [*Note:* The standard containers (ISO/IEC 14882:2014 §23) satisfy `BoundedRange`. — *end note*]

```
template <class T>
concept bool BoundedRange =
  Range<T> && Same<iterator_t<T>, sentinel_t<T>>;
```

10.6.6 Input ranges [ranges.input]

1 The `InputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that satisfies `InputIterator` (9.3.11).

```
template <class T>
concept bool InputRange =
  Range<T> && InputIterator<iterator_t<T>>;
```

10.6.7 Output ranges [ranges.output]

1 The `OutputRange` concept specifies requirements of a `Range` type for which `begin` returns a type that satisfies `OutputIterator` (9.3.12).

```
template <class R, class T>
concept bool OutputRange =
  Range<R> && OutputIterator<iterator_t<R>, T>;
```

10.6.8 Forward ranges [ranges.forward]

1 The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that satisfies `ForwardIterator` (9.3.13).

```
template <class T>
concept bool ForwardRange =
  InputRange<T> && ForwardIterator<iterator_t<T>>;
```

10.6.9 Bidirectional ranges [ranges.bidirectional]

1 The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that satisfies `BidirectionalIterator` (9.3.14).

```
template <class T>
concept bool BidirectionalRange =
  ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;
```

10.6.10 Random access ranges [ranges.random.access]

1 The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that satisfies `RandomAccessIterator` (9.3.15).

```
template <class T>
concept bool RandomAccessRange =
  BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```

11 Algorithms library [algorithms]

11.1 General [algorithms.general]

1 This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause ISO/IEC 14882:2014 §23) and other sequences.

2 The following subclauses describe components for non-modifying sequence operations, modifying sequence operations, and sorting and related operations, as summarized in Table 11.

Table 11 — Algorithms library summary

	Subclause	Header(s)
11.3	Non-modifying sequence operations	
11.4	Mutating sequence operations	`<experimental/ranges/algorithm>`
11.5	Sorting and related operations	

Header `<experimental/ranges/algorithm>` synopsis

```
#include <initializer_list>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  namespace tag {
    // 11.2, tag specifiers (See 8.5.2):
    struct in;
    struct in1;
    struct in2;
    struct out;
    struct out1;
    struct out2;
    struct fun;
    struct min;
    struct max;
    struct begin;
    struct end;
  }

  // 11.3, non-modifying sequence operations:
  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});

  template <InputIterator I, Sentinel<I> S, class Proj = identity,
      IndirectUnaryPredicate<projected<I, Proj>> Pred>
    bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

  template <InputRange Rng, class Proj = identity,
      IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
```

```
  bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryInvocable<projected<I, Proj>> Fun>
  tagged_pair<tag::in(I), tag::fun(Fun)>
    for_each(I first, S last, Fun f, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    for_each(Rng&& rng, Fun f, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I find(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Proj = identity,
    IndirectRelation<I2, projected<I1, Proj>> Pred = equal_to<>>
  I1
    find_end(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{}, Proj proj = Proj{});

template <ForwardRange Rng1, ForwardRange Rng2, class Proj = identity,
```

```
    IndirectRelation<iterator_t<Rng2>,
      projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  I1
    find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
                  Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, ForwardRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_first_of(Rng1&& rng1, Rng2&& rng2,
                  Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> Pred = equal_to<>>
  I
    adjacent_find(I first, S last, Pred pred = Pred{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng>
    adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  difference_type_t<I>
    count(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  difference_type_t<iterator_t<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  difference_type_t<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
```

```
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>),
              tag::in2(safe_iterator_t<Rng2>)>
    mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});


template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
    class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                      Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  I1
    search(I1 first1, S1 last1, I2 first2, S2 last2,
           Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  safe_iterator_t<Rng1>
```

```
    search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class T,
    class Pred = equal_to<>, class Proj = identity>
  requires IndirectlyComparable<I, const T*, Pred, Proj>
  I
    search_n(I first, S last, difference_type_t<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
  safe_iterator_t<Rng>
    search_n(Rng&& rng, difference_type_t<iterator_t<Rng>> count,
             const T& value, Pred pred = Pred{}, Proj proj = Proj{});

// 11.4, modifying sequence operations:
// 11.4.1, copy:
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy(Rng&& rng, O result);

template <InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, difference_type_t<I> n, O result);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});

template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    copy_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
```

```
    copy_backward(Rng&& rng, I result);

// 11.4.2, move:
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    move(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyMovable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    move(Rng&& rng, O result);

template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    move_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    move_backward(Rng&& rng, I result);

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
  requires IndirectlySwappable<I1, I2>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template <ForwardRange Rng1, ForwardRange Rng2>
  requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    swap_ranges(Rng1&& rng1, Rng2&& rng2);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    CopyConstructible F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>
  tagged_pair<tag::in(I), tag::out(O)>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
    class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<R>, Proj>)>>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    transform(Rng&& rng, O result, F op, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
    class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I1, Proj1>,
    projected<I2, Proj2>)>>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    transform(Rng1&& rng1, Rng2&& rng2, O result,
              F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
  requires Writable<I, const T2&> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  I
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, class Proj = identity>
  requires Writable<iterator_t<Rng>, const T2&> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  safe_iterator_t<Rng>
    replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Writable<I, const T&>
  I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Writable<iterator_t<Rng>, const T&>
  safe_iterator_t<Rng>
    replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
```

```
                    Proj proj = Proj{});

template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <class T, OutputIterator<const T&> O, Sentinel<O> S>
  O fill(O first, S last, const T& value);

template <class T, OutputRange<const T&> Rng>
  safe_iterator_t<Rng>
    fill(Rng&& rng, const T& value);

template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);

template <Iterator O, Sentinel<O> S, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate(O first, S last, F gen);

template <class Rng, CopyConstructible F>
    requires Invocable<F&> && OutputRange<Rng, result_of_t<F&()>>
  safe_iterator_t<Rng>
    generate(Rng&& rng, F gen);

template <Iterator O, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate_n(O first, difference_type_t<O> n, F gen);

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires Permutable<I> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I remove(I first, S last, const T& value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity>
  requires Permutable<iterator_t<Rng>> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    remove(Rng&& rng, const T& value, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I remove_if(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
```

```
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires Permutable<I>
  I unique(I first, S last, R comp = R{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<I, O> &&
    (ForwardIterator<I> ||
     (InputIterator<O> && Same<value_type_t<I>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<I, O>)
  tagged_pair<tag::in(I), tag::out(O)>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    (ForwardIterator<iterator_t<Rng>> ||
     (InputIterator<O> && Same<value_type_t<iterator_t<Rng>>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<iterator_t<Rng>, O>)
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});

template <BidirectionalIterator I, Sentinel<I> S>
  requires Permutable<I>
```

```
  I reverse(I first, S last);

template <BidirectionalRange Rng>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    reverse(Rng&& rng);

template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);

template <BidirectionalRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    reverse_copy(Rng&& rng, O result);

template <ForwardIterator I, Sentinel<I> S>
  requires Permutable<I>
  tagged_pair<tag::begin(I), tag::end(I)>
    rotate(I first, I middle, S last);

template <ForwardRange Rng>
  requires Permutable<iterator_t<Rng>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    rotate(Rng&& rng, iterator_t<Rng> middle);

template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    rotate_copy(I first, I middle, S last, O result);

template <ForwardRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);

// 11.4.12, shuffle:
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  I shuffle(I first, S last, Gen&& g);

template <RandomAccessRange Rng, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  safe_iterator_t<Rng>
    shuffle(Rng&& rng, Gen&& g);

// 11.4.13, partitions:
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});
```

```
template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool
    is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I partition(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    partition(Rng&& rng, Pred pred, Proj proj = Proj{});

template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});

template <BidirectionalRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
  tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
    partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O1> &&
    IndirectlyCopyable<iterator_t<Rng>, O2>
  tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
    partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});

// 11.5, sorting and related operations:
// 11.5.1, sorting:
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
```

```
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
  I2
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                      Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, RandomAccessRange Rng2, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
      Sortable<iterator_t<Rng2>, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>>
  safe_iterator_t<Rng2>
    partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
```

```
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});

// 11.5.3, binary search:
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::begin(I), tag::end(I)>
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
```

```
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  bool
    binary_search(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

// 11.5.4, merge:
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    merge(Rng1&& rng1, Rng2&& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});

// 11.5.5, set operations:
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
  bool
    includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
```

```
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  O
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  O
    set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(I1), tag::out(O)>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
```

```
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 11.5.6, heap operations:
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
```

```
  bool
    is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

// 11.5.7, minimum and maximum:
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T min(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T max(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(T), tag::max(T)>
    minmax(initializer_list<T> t, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
```

```
                tag::max(value_type_t<iterator_t<Rng>>)>
      minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
    I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    safe_iterator_t<Rng>
      min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
    I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    safe_iterator_t<Rng>
      max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
      IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
    tagged_pair<tag::min(I), tag::max(I)>
      minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardRange Rng, class Proj = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
    tagged_pair<tag::min(safe_iterator_t<Rng>),
                tag::max(safe_iterator_t<Rng>)>
      minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
      class Proj1 = identity, class Proj2 = identity,
      IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
    bool
      lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                              Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
      class Proj2 = identity,
      IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
    bool
      lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

  // 11.5.9, permutations:
  template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = less<>,
      class Proj = identity>
```

```
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    bool
      next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
      class Proj = identity>
    requires Sortable<I, Comp, Proj>
    bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

  template <BidirectionalRange Rng, class Comp = less<>,
      class Proj = identity>
    requires Sortable<iterator_t<Rng>, Comp, Proj>
    bool
      prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
}}}}
```

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.

5 Both in-place and copying versions are provided for certain algorithms.[4] When such a version is provided for *algorithm* it is called *algorithm_copy*. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

6 [*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>` (ISO/IEC 14882:2014 §20.9.3), or some equivalent solution. — *end note*]

7 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```
X tmp = a;
advance(tmp, n);
return tmp;
```

and that of `b-a` is the same as of

```
return distance(a, b);
```

8 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;
```

9 Overloads of algorithms that take `Range` arguments (10.6.2) behave as if they are implemented by calling `begin` and `end` on the `Range` and dispatching to the overload that takes separate iterator and sentinel arguments.

4) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

10 The number and order of template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise.

11.2 Tag specifiers [alg.tagspec]

```
namespace tag {
  struct in { /* implementation-defined */ };
  struct in1 { /* implementation-defined */ };
  struct in2 { /* implementation-defined */ };
  struct out { /* implementation-defined */ };
  struct out1 { /* implementation-defined */ };
  struct out2 { /* implementation-defined */ };
  struct fun { /* implementation-defined */ };
  struct min { /* implementation-defined */ };
  struct max { /* implementation-defined */ };
  struct begin { /* implementation-defined */ };
  struct end { /* implementation-defined */ };
}
```

1 In the following description, let *X* be the name of a type in the `tag` namespace above.

2 `tag::`*X* is a tag specifier (8.5.2) such that *`TAGGET`*`(`*D*`, tag::`*X*`, `*N*`)` names a tagged getter (8.5.2) with DerivedCharacteristic *D*, ElementIndex *N*, and ElementName *X*.

3 [*Example:* `tag::in` is a type such that *`TAGGET`*`(`*D*`, tag::in, `*N*`)` names a type with the following interface:

```
struct __input_getter {
  constexpr decltype(auto) in() &       { return get<N>(static_cast<D&>(*this)); }
  constexpr decltype(auto) in() &&      { return get<N>(static_cast<D&&>(*this)); }
  constexpr decltype(auto) in() const & { return get<N>(static_cast<const D&>(*this)); }
};
```

— *end example*]

11.3 Non-modifying sequence operations [alg.nonmodifying]

11.3.1 All of [alg.all_of]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool all_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if `invoke(pred, invoke(proj, *i))` is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

11.3.2 Any of [alg.any_of]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool any_of(I first, S last, Pred pred, Proj proj = Proj{});
```

```
template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool any_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `false` if `[first,last)` is empty or if there is no iterator `i` in the range `[first,last)` such that `invoke(pred, invoke(proj, *i))` is `true`, and `true` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

11.3.3 None of [alg.none_of]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool none_of(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if `invoke(pred, invoke(proj, *i))` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate and `last - first` applications of the projection.

11.3.4 For each [alg.foreach]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryInvocable<projected<I, Proj>> Fun>
  tagged_pair<tag::in(I), tag::fun(Fun)>
    for_each(I first, S last, Fun f, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
    for_each(Rng&& rng, Fun f, Proj proj = Proj{});
```

1 *Effects:* Calls `invoke(f, invoke(proj, *i))` for every iterator `i` in the range `[first,last)`, starting from `first` and proceeding to `last - 1`. [*Note:* If the result of `invoke(proj, *i)` is a mutable reference, `f` may apply nonconstant functions. — *end note*]

2 *Returns:* `{last, std::move(f)}`.

3 *Complexity:* Applies `f` and `proj` exactly `last - first` times.

4 *Remarks:* If `f` returns a result, the result is ignored.

5 [*Note:* The requirements of this algorithm are more strict than those specified in ISO/IEC 14882:2014 §25.2.4. This algorithm requires `Fun` to satisfy `CopyConstructible`, whereas the algorithm in the C++ Standard requires only `MoveConstructible`. — *end note*]

11.3.5 Find [alg.find]

```
template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  I find(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
```

```
  safe_iterator_t<Rng>
    find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if(Rng&& rng, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    find_if_not(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`, `invoke(pred, invoke(proj, *i)) == false`. Returns `last` if no such iterator is found.

2 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

11.3.6 Find end [alg.find.end]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Proj = identity,
    IndirectRelation<I2, projected<I1, Proj>> Pred = equal_to<>>
  I1
    find_end(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{}, Proj proj = Proj{});

template <ForwardRange Rng1, ForwardRange Rng2,
    class Proj = identity,
    IndirectRelation<iterator_t<Rng2>,
      projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_end(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, Proj proj = Proj{});
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for every non-negative integer `n < (last2 - first2)`, the following condition holds: `invoke(pred, invoke(proj, *(i + n)), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate and projection.

11.3.7 Find first of [alg.find.first.of]

```
template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
```

```
  I1
    find_first_of(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, ForwardRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  safe_iterator_t<Rng1>
    find_first_of(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                  Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Finds an element that matches one of a set of values.

2 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following condition holds: `invoke(pred, invoke(proj1, *i), invoke(proj2, *j)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.

3 *Complexity:* At most `(last1-first1) * (last2-first2)` applications of the corresponding predicate and the two projections.

11.3.8 Adjacent find [alg.adjacent.find]

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> Pred = equal_to<>>
  I
    adjacent_find(I first, S last, Pred pred = Pred{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> Pred = equal_to<>>
  safe_iterator_t<Rng>
    adjacent_find(Rng&& rng, Pred pred = Pred{}, Proj proj = Proj{});
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding condition holds: `invoke(pred, invoke(proj, *i), invoke(proj, *(i + 1))) != false`. Returns `last` if no such iterator is found.

2 *Complexity:* For a nonempty range, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value, and no more than twice as many applications of the projection.

11.3.9 Count [alg.count]

```
template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  difference_type_t<I>
    count(I first, S last, const T& value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity>
  requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  difference_type_t<iterator_t<Rng>>
    count(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  difference_type_t<I>
    count_if(I first, S last, Pred pred, Proj proj = Proj{});
```

```
template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

2 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

11.3.10 Mismatch [mismatch]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Pred = equal_to<>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    mismatch(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* A pair of iterators `i` and `j` such that `j == first2 + (i - first1)` and `i` is the first iterator in the range `[first1,last1)` for which the following corresponding conditions hold:

(1.1) — `j` is in the range `[first2, last2)`.

(1.2) — `!(*i == *(first2 + (i - first1)))`

(1.3) — `invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1)))) == 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.

2 *Complexity:* At most `last1 - first1` applications of the corresponding predicate and both projections.

11.3.11 Equal [alg.equal]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
             Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if for every iterator `i` in the range `[first1,last1)` the following condition holds: `invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1)))) != false`. Otherwise, returns `false`.

2 *Complexity:* No applications of the corresponding predicate and projections if `SizedSentinel<S1, I1>` is satisfied, and `SizedSentinel<S2, I2>` is satisfied, and `last1 - first1 != last2 - first2`. Otherwise, at most `min(last1 - first1, last2 - first2)` applications of the corresponding predicate and projections.

11.3.12 Is permutation [alg.is_permutation]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
    class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                      Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* If `last1 - first1 != last2 - first2`, return `false`. Otherwise return `true` if there exists a permutation of the elements in the range `[first2,first2 + (last1 - first1))`, beginning with `I2 begin`, such that `equal(first1, last1, begin, pred, proj1, proj2)` returns `true` ; otherwise, returns `false`.

2 *Complexity:* No applications of the corresponding predicate and projections if `SizedSentinel<S1, I1>` is satisfied, and `SizedSentinel<S2, I2>` is satisfied, and `last1 - first1 != last2 - first2`. Otherwise, exactly `last1 - first1` applications of the corresponding predicate and projections if `equal(first1, last1, first2, last2, pred, proj1, proj2)` would return `true`; otherwise, at worst $\mathcal{O}(N^2)$, where N has the value `last1 - first1`.

11.3.13 Search [alg.search]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  I1
    search(I1 first1, S1 last1, I2 first2, S2 last2,
           Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>
  safe_iterator_t<Rng1>
    search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
           Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for every non-negative integer `n` less than `last2 - first2` the following condition holds:

```
invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) != false.
```

Returns `first1` if `[first2,last2)` is empty, otherwise returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate and projections.

```
template <ForwardIterator I, Sentinel<I> S, class T,
    class Pred = equal_to<>, class Proj = identity>
  requires IndirectlyComparable<I, const T*, Pred, Proj>
  I
    search_n(I first, S last, difference_type_t<I> count,
             const T& value, Pred pred = Pred{},
             Proj proj = Proj{});

template <ForwardRange Rng, class T, class Pred = equal_to<>,
    class Proj = identity>
  requires IndirectlyComparable<iterator_t<Rng>, const T*, Pred, Proj>
  safe_iterator_t<Rng>
    search_n(Rng&& rng, difference_type_t<iterator_t<Rng>> count,
             const T& value, Pred pred = Pred{}, Proj proj = Proj{});
```

4 *Effects:* Finds a subsequence of equal values in a sequence.

5 *Returns:* The first iterator `i` in the range `[first,last-count)` such that for every non-negative integer `n` less than `count` the following condition holds: `invoke(pred, invoke(proj, *(i + n)), value) != false`. Returns `last` if no such iterator is found.

6 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

11.4 Mutating sequence operations [alg.modifying.operations]

11.4.1 Copy [alg.copy]

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy(Rng&& rng, O result);
```

1 *Effects:* Copies elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last - first)`, performs `*(result + n) = *(first + n)`.

2 *Returns:* `{last, result + (last - first)}`.

3 *Requires:* `result` shall not be in the range `[first,last)`.

4 *Complexity:* Exactly `last - first` assignments.

```
template <InputIterator I, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_n(I first, difference_type_t<I> n, O result);
```

5 *Effects:* For each non-negative integer $i < n$, performs `*(result + i) = *(first + i)`.

6 *Returns:* `{first + n, result + n}`.

7 *Complexity:* Exactly `n` assignments.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

8 Let N be the number of iterators `i` in the range `[first,last)` for which the condition `invoke(pred, invoke(proj, *i))` holds.

9 *Requires:* The ranges `[first,last)` and `[result,result +` N`)` shall not overlap.

10 *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `invoke(pred, invoke(proj, *i))` is `true`.

11 *Returns:* `{last, result +` N`}`.

12 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

13 *Remarks:* Stable (ISO/IEC 14882:2014 §17.6.5.7).

```
template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyCopyable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    copy_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyCopyable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    copy_backward(Rng&& rng, I result);
```

14 *Effects:* Copies elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to `first`.[5] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

15 *Requires:* `result` shall not be in the range `(first,last]`.

16 *Returns:* `{last, result - (last - first)}`.

17 *Complexity:* Exactly `last - first` assignments.

11.4.2 Move [alg.move]

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyMovable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    move(I first, S last, O result);

template <InputRange Rng, WeaklyIncrementable O>
```

5) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

```
  requires IndirectlyMovable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    move(Rng&& rng, O result);
```

1 *Effects:* Moves elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from first and proceeding to last. For each non-negative integer `n < (last-first)`, performs `*(result + n) = ranges::iter_move(first + n)`.

2 *Returns:* `{last, result + (last - first)}`.

3 *Requires:* `result` shall not be in the range `[first,last)`.

4 *Complexity:* Exactly `last - first` move assignments.

```
template <BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
  requires IndirectlyMovable<I1, I2>
  tagged_pair<tag::in(I1), tag::out(I2)>
    move_backward(I1 first, S1 last, I2 result);

template <BidirectionalRange Rng, BidirectionalIterator I>
  requires IndirectlyMovable<iterator_t<Rng>, I>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(I)>
    move_backward(Rng&& rng, I result);
```

5 *Effects:* Moves elements in the range `[first,last)` into the range `[result - (last-first),result )` starting from `last - 1` and proceeding to first.[6] For each positive integer `n <= (last - first)`, performs `*(result - n) = ranges::iter_move(last - n)`.

6 *Requires:* `result` shall not be in the range `(first,last]`.

7 *Returns:* `{last, result - (last - first)}`.

8 *Complexity:* Exactly `last - first` assignments.

11.4.3 swap [alg.swap]

```
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
  requires IndirectlySwappable<I1, I2>
  tagged_pair<tag::in1(I1), tag::in2(I2)>
    swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template <ForwardRange Rng1, ForwardRange Rng2>
  requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
    swap_ranges(Rng1&& rng1, Rng2&& rng2);
```

1 *Effects:* For each non-negative integer `n < min(last1 - first1, last2 - first2)` performs: `ranges::iter_swap(first1 + n, first2 + n)`.

2 *Requires:* The two ranges `[first1,last1)` and `[first2,last2)` shall not overlap. `*(first1 + n)` shall be swappable with (7.3.11) `*(first2 + n)`.

3 *Returns:* `{first1 + n, first2 + n}`, where `n` is `min(last1 - first1, last2 - first2)`.

4 *Complexity:* Exactly `min(last1 - first1, last2 - first2)` swaps.

11.4.4 Transform [alg.transform]

6) `move_backward` should be used instead of move when last is in the range `[result - (last - first),result)`.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    CopyConstructible F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>
  tagged_pair<tag::in(I), tag::out(O)>
    transform(I first, S last, O result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, CopyConstructible F,
    class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<R>, Proj>)>>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    transform(Rng&& rng, O result, F op, Proj proj = Proj{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
    class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I1, Proj1>,
    projected<I2, Proj2>)>>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
            F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    transform(Rng1&& rng1, Rng2&& rng2, O result,
             F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 Let N be `(last1 - first1)` for unary transforms, or `min(last1 - first1, last2 - first2) for binary transforms.`

2 *Effects:* Assigns through every iterator `i` in the range `[result,result +` N`)` a new corresponding value equal to `invoke(op, invoke(proj, *(first1 + (i - result))))` or `invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))`.

3 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,first1 +` N`]`, `[first2,first2 +` N`]`, and `[result,result +` N`]`.[7]

4 *Returns:* `{first1 +` N`, result +` N`}` or `make_tagged_tuple<tag::in1, tag::in2, tag::out>( first1 +` N`, first2 +` N`, result +` N`)`.

5 *Complexity:* Exactly N applications of `op` or `binary_op` and the corresponding projection(s).

6 *Remarks:* `result` may be equal to `first1` in case of unary transform, or to `first1` or `first2` in case of binary transform.

11.4.5 Replace [alg.replace]

```
template <InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
  requires Writable<I, const T2&> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  I
```

7) The use of fully closed ranges is intentional.

```
    replace(I first, S last, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, class Proj = identity>
  requires Writable<iterator_t<Rng>, const T2&> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  safe_iterator_t<Rng>
    replace(Rng&& rng, const T1& old_value, const T2& new_value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Writable<I, const T&>
  I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <InputRange Rng, class T, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Writable<iterator_t<Rng>, const T&>
  safe_iterator_t<Rng>
    replace_if(Rng&& rng, Pred pred, const T& new_value, Proj proj = Proj{});
```

1 *Effects:* Assigns `new_value` through each iterator `i` in the range `[first,last)` when the following corresponding conditions hold: `invoke(proj, *i) == old_value`, `invoke(pred, invoke(proj, *i)) != false`.

2 *Returns:* `last`.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

```
template <InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T1*>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputRange Rng, class T1, class T2, OutputIterator<const T2&> O,
    class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T1*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy(Rng&& rng, O result, const T1& old_value, const T2& new_value,
                 Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});

template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    replace_copy_if(Rng&& rng, O result, Pred pred, const T& new_value,
                    Proj proj = Proj{});
```

4 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

5 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
invoke(proj, *(first + (i - result))) == old_value
invoke(pred, invoke(proj, *(first + (i - result)))) != false
```

6 *Returns:* `{last, result + (last - first)}`.

7 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

11.4.6 Fill [alg.fill]

```
template <class T, OutputIterator<const T&> O, Sentinel<O> S>
  O fill(O first, S last, const T& value);

template <class T, OutputRange<const T&> Rng>
  safe_iterator_t<Rng>
    fill(Rng&& rng, const T& value);

template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);
```

1 *Effects:* `fill` assigns `value` through all the iterators in the range `[first,last)`. `fill_n` assigns `value` through all the iterators in the counted range `[first,n)` if `n` is positive, otherwise it does nothing.

2 *Returns:* `last`, where `last` is `first + max(n, 0)` for `fill_n`.

3 *Complexity:* Exactly `last - first` assignments.

11.4.7 Generate [alg.generate]

```
template <Iterator O, Sentinel<O> S, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate(O first, S last, F gen);

template <class Rng, CopyConstructible F>
    requires Invocable<F&> && OutputRange<Rng, result_of_t<F&()>>
  safe_iterator_t<Rng>
    generate(Rng&& rng, F gen);

template <Iterator O, CopyConstructible F>
    requires Invocable<F&> && Writable<O, result_of_t<F&()>>
  O generate_n(O first, difference_type_t<O> n, F gen);
```

1 *Effects:* The generate algorithms invoke the function object `gen` and assign the return value of `gen` through all the iterators in the range `[first,last)`. The `generate_n` algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the counted range `[first,n)` if `n` is positive, otherwise it does nothing.

2 *Returns:* `last`, where `last` is `first + max(n, 0)` for `generate_n`.

3 *Complexity:* Exactly `last - first` evaluations of `invoke(gen)` and assignments.

11.4.8 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 <ForwardRange Rng, class T, class Proj = identity>
  requires Permutable<iterator_t<Rng>> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  safe_iterator_t<Rng>
    remove(Rng&& rng, const T& value, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I remove_if(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    remove_if(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

2 *Returns:* The end of the resulting range.

3 *Remarks:* Stable (ISO/IEC 14882:2014 §17.6.5.7).

4 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

5 *Note:* each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
    class Proj = identity>
  requires IndirectlyCopyable<I, O> &&
    IndirectRelation<equal_to<>, projected<I, Proj>, const T*>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy(I first, S last, O result, const T& value, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class T, class Proj = identity>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    IndirectRelation<equal_to<>, projected<iterator_t<Rng>, Proj>, const T*>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy(Rng&& rng, O result, const T& value, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    remove_copy_if(I first, S last, O result, Pred pred, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    remove_copy_if(Rng&& rng, O result, Pred pred, Proj proj = Proj{});
```

6 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

7 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `invoke(proj, *i) == value`, `invoke(pred, invoke(proj, *i)) != false`.

8 *Returns:* A pair consisting of `last` and the end of the resulting range.

9 *Complexity:* Exactly `last - first` applications of the corresponding predicate and projection.

10 *Remarks:* Stable (ISO/IEC 14882:2014 §17.6.5.7).

11.4.9 Unique [alg.unique]

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires Permutable<I>
  I unique(I first, S last, R comp = R{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    unique(Rng&& rng, R comp = R{}, Proj proj = Proj{});
```

1 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `invoke(proj, *(i - 1)) == invoke(proj, *i)` or `invoke(pred, invoke(proj, *(i - 1)), invoke(proj, *i)) != false`.

2 *Returns:* The end of the resulting range.

3 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    class Proj = identity, IndirectRelation<projected<I, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<I, O> &&
    (ForwardIterator<I> ||
     (InputIterator<O> && Same<value_type_t<I>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<I, O>)
  tagged_pair<tag::in(I), tag::out(O)>
    unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
    IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
  requires IndirectlyCopyable<iterator_t<Rng>, O> &&
    (ForwardIterator<iterator_t<Rng>> ||
     (InputIterator<O> && Same<value_type_t<iterator_t<Rng>>, value_type_t<O>>) ||
     IndirectlyCopyableStorable<iterator_t<Rng>, O>)
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    unique_copy(Rng&& rng, O result, R comp = R{}, Proj proj = Proj{});
```

4 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

5 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold:

```
invoke(proj, *i) == invoke(proj, *(i - 1))
```

or

`invoke(pred, invoke(proj, *i), invoke(proj, *(i - 1))) != false`.

6 *Returns:* A pair consisting of `last` and the end of the resulting range.

7 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate and no more than twice as many applications of the projection.

11.4.10 Reverse [alg.reverse]

```
template <BidirectionalIterator I, Sentinel<I> S>
  requires Permutable<I>
  I reverse(I first, S last);

template <BidirectionalRange Rng>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    reverse(Rng&& rng);
```

1 *Effects:* For each non-negative integer `i < (last - first)/2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

2 *Returns:* `last`.

3 *Complexity:* Exactly `(last - first)/2` swaps.

```
template <BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)> reverse_copy(I first, S last, O result);

template <BidirectionalRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    reverse_copy(Rng&& rng, O result);
```

4 *Effects:* Copies the range `[first,last)` to the range `[result,result+(last-first))` such that for every non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - 1 - i) = *(first + i)`.

5 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

6 *Returns:* `{last, result + (last - first)}`.

7 *Complexity:* Exactly `last - first` assignments.

11.4.11 Rotate [alg.rotate]

```
template <ForwardIterator I, Sentinel<I> S>
  requires Permutable<I>
  tagged_pair<tag::begin(I), tag::end(I)> rotate(I first, I middle, S last);

template <ForwardRange Rng>
  requires Permutable<iterator_t<Rng>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>), tag::end(safe_iterator_t<Rng>)>
    rotate(Rng&& rng, iterator_t<Rng> middle);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* `{first + (last - middle), last}`.

3 *Remarks:* This is a left rotate.

4 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges.

5 *Complexity:* At most `last - first` swaps.

```
template <ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  tagged_pair<tag::in(I), tag::out(O)>
    rotate_copy(I first, I middle, S last, O result);

template <ForwardRange Rng, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<Rng>, O>
  tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
    rotate_copy(Rng&& rng, iterator_t<Rng> middle, O result);
```

6 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

7 *Returns:* `{last, result + (last - first)}`.

8 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

9 *Complexity:* Exactly `last - first` assignments.

11.4.12 Shuffle [alg.random.shuffle]

```
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  I shuffle(I first, S last, Gen&& g);

template <RandomAccessRange Rng, class Gen>
  requires Permutable<I> &&
    UniformRandomNumberGenerator<remove_reference_t<Gen>> &&
    ConvertibleTo<result_of_t<Gen&()>, difference_type_t<I>>
  safe_iterator_t<Rng>
    shuffle(Rng&& rng, Gen&& g);
```

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 *Complexity:* Exactly `(last - first) - 1` swaps.

3 *Returns:* `last`

4 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

11.4.13 Partitions [alg.partitions]

```
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  bool
    is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1 *Returns:* `true` if `[first,last)` is empty or if `[first,last)` is partitioned by `pred` and `proj`, i.e. if all iterators `i` for which `invoke(pred, invoke(proj, *i)) != false` come before those that do not, for every `i` in `[first,last)`.

2 *Complexity:* Linear. At most `last - first` applications of `pred` and `proj`.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I partition(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

3 *Effects:* Permutes the elements in the range `[first,last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

4 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

5 *Complexity:* If `I` meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps; otherwise at most `last - first` swaps. Exactly `last - first` applications of the predicate and projection.

```
template <BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<I>
  I stable_partition(I first, S last, Pred pred, Proj proj = Proj{});

template <BidirectionalRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires Permutable<iterator_t<Rng>>
  safe_iterator_t<Rng>
    stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

6 *Effects:* Permutes the elements in the range `[first,last)` such that there exists an iterator `i` such that for every iterator `j` in the range `[first,i)` `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`.

7 *Returns:* An iterator `i` such that for every iterator `j` in the range `[first,i)`, `invoke(pred, invoke(proj, *j)) != false`, and for every iterator `k` in the range `[i,last)`, `invoke(pred, invoke(proj, *k)) == false`. The relative order of the elements in both groups is preserved.

8 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate and projection.

```
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
  tagged_tuple<tag::in(I), tag::out1(O1), tag::out2(O2)>
    partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred,
                   Proj proj = Proj{});
```

```
template <InputRange Rng, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  requires IndirectlyCopyable<iterator_t<Rng>, O1> &&
    IndirectlyCopyable<iterator_t<Rng>, O2>
  tagged_tuple<tag::in(safe_iterator_t<Rng>), tag::out1(O1), tag::out2(O2)>
    partition_copy(Rng&& rng, O1 out_true, O2 out_false, Pred pred, Proj proj = Proj{});
```

9 *Requires:* The input range shall not overlap with either of the output ranges.

10 *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `invoke(pred, invoke(proj, *i))` is `true`, or to the output range beginning with `out_false` otherwise.

11 *Returns:* A tuple `p` such that `get<0>(p)` is `last`, `get<1>(p)` is the end of the output range beginning at `out_true`, and `get<2>(p)` is the end of the output range beginning at `out_false`.

12 *Complexity:* Exactly `last - first` applications of `pred` and `proj`.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
  I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<Rng>, Proj>> Pred>
  safe_iterator_t<Rng>
    partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});
```

13 *Requires:* `[first,last)` shall be partitioned by `pred` and `proj`, i.e. there shall be an iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

14 *Returns:* An iterator `mid` such that `all_of(first, mid, pred, proj)` and `none_of(mid, last, pred, proj)` are both true.

15 *Complexity:* $\mathcal{O}(\log($`last - first`$))$ applications of `pred` and `proj`.

11.5 Sorting and related operations [alg.sorting]

1 All the operations in 11.5 take an optional binary callable predicate of type `Comp` that defaults to `less<>`.

2 `Comp` is a callable object (ISO/IEC 14882:2014 §20.9.2). The return value of the `invoke` operation applied to an object of type `Comp`, when contextually converted to `bool` (Clause ISO/IEC 14882:2014 §4), yields `true` if the first argument of the call is less than the second, and `false` otherwise. `Comp comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

3 A sequence is *sorted with respect to a comparator and projection* `comp` and `proj` if for every iterator `i` pointing to the sequence and every non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `invoke(comp, invoke(proj, *(i + n)), invoke(proj, *i)) == false`.

4 A sequence `[start,finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < distance(start, finish)`, `f(*(start + i))` is true if and only if `i < n`.

5 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

11.5.1 Sorting [alg.sort]

11.5.1.1 `sort` [sort]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Returns:* `last`.

3 *Complexity:* $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons, and twice as many applications of the projection.

11.5.1.2 `stable_sort` [stable.sort]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Returns:* `last`.

3 *Complexity:* Let N `== last - first`. If enough extra memory is available, $N \log(N)$ comparisons. Otherwise, at most $N \log^2(N)$ comparisons. In either case, twice as many applications of the projection as the number of comparisons.

4 *Remarks:* Stable (ISO/IEC 14882:2014 §17.6.5.7).

11.5.1.3 `partial_sort` [partial.sort]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I partial_sort(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    partial_sort(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                 Proj proj = Proj{});
```

1 *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

2 *Returns:* `last`.

3 *Complexity:* It takes approximately `(last - first) * log(middle - first)` comparisons, and exactly twice as many applications of the projection.

11.5.1.4 `partial_sort_copy` [partial.sort.copy]

```
template <InputIterator I1, Sentinel<I1> S1, RandomAccessIterator I2, Sentinel<I2> S2,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<I1, I2> && Sortable<I2, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<I1, Proj1>, projected<I2, Proj2>>
  I2
    partial_sort_copy(I1 first, S1 last, I2 result_first, S2 result_last,
                      Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, RandomAccessRange Rng2, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyCopyable<iterator_t<Rng1>, iterator_t<Rng2>> &&
      Sortable<iterator_t<Rng2>, Comp, Proj2> &&
      IndirectStrictWeakOrder<Comp, projected<iterator_t<Rng1>, Proj1>,
        projected<iterator_t<Rng2>, Proj2>>
  safe_iterator_t<Rng2>
    partial_sort_copy(Rng1&& rng, Rng2&& result_rng, Comp comp = Comp{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

2 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

3 *Complexity:* Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons, and exactly twice as many applications of the projection.

11.5.1.5 `is_sorted` [is.sorted]

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* `is_sorted_until(first, last, comp, proj) == last`

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

2 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is sorted.

3 *Complexity:* Linear.

11.5.2 Nth element [alg.nth.element]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
```

```
  requires Sortable<I, Comp, Proj>
  I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});
```

1 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted, unless `nth == last`. Also for every iterator `i` in the range `[first,nth)` and every iterator `j` in the range `[nth,last)` it holds that: `invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`.

2 *Returns:* `last`.

3 *Complexity:* Linear on average.

11.5.3 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 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.

11.5.3.1 `lower_bound` [lower.bound]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    lower_bound(I first, S last, const T& value, Comp comp = Comp{},
                Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    lower_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `invoke(comp, invoke(proj, e), value)`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding condition holds: `invoke(comp, invoke(proj, *j), value) != false`.

3 *Complexity:* At most $\log_2$(`last - first`) $+ \mathcal{O}(1)$ applications of the comparison function and projection.

11.5.3.2 `upper_bound` [upper.bound]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  I
    upper_bound(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
```

```
upper_bound(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for every iterator `j` in the range `[first,i)` the following corresponding condition holds: `invoke(comp, value, invoke(proj, *j)) == false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

11.5.3.3 `equal_range` [equal.range]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::begin(I), tag::end(I)>
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:*

```
{lower_bound(first, last, value, comp, proj),
 upper_bound(first, last, value, comp, proj)}
```

3 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

11.5.3.4 `binary_search` [binary.search]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  bool
    binary_search(I first, S last, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    binary_search(Rng&& rng, const T& value, Comp comp = Comp{},
                  Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `invoke(comp, invoke(proj, *i), value) == false && invoke(comp, value, invoke(proj, *i)) == false`.

3 *Complexity:* At most $\log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

11.5.4 Merge [alg.merge]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity,
    class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    merge(I1 first1, S1 last1, I2 first2, S2 last2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class Comp = less<>,
    class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    merge(Rng1&& rng1, Rng2&& rng2, O result,
          Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`. If an element `a` precedes `b` in an input range, `a` is copied into the output range before `b`. If `e1` is an element of `[first1,last1)` and `e2` of `[first2,last2)`, `e2` is copied into the output range before `e1` if and only if `bool(invoke(comp, invoke(proj2, e2), invoke(proj1, e1)))` is `true`.

2 *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `comp`, `proj1`, and `proj2`. The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result_last)`.

4 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` applications of the comparison function and each projection.

5 *Remarks:* Stable (ISO/IEC 14882:2014 §17.6.5.7).

```
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I
    inplace_merge(I first, I middle, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    inplace_merge(Rng&& rng, iterator_t<Rng> middle, Comp comp = Comp{},
                  Proj proj = Proj{});
```

6 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))` will be false.

7 *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to `comp` and `proj`.

8 *Returns:* `last`

9 *Complexity:* When enough additional memory is available, `(last - first) - 1` applications of the comparison function and projection. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

10 *Remarks:* Stable (ISO/IEC 14882:2014 §17.6.5.7).

11.5.5 Set operations on sorted structures [alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They also work with `multiset`s (ISO/IEC 14882:2014 §23.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multiset`s in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

11.5.5.1 `includes` [includes]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
  bool
    includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

11.5.5.2 `set_union` [set.union]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_union(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
              Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result +` n`)`, where n is the number of elements in the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

11.5.5.3 `set_intersection` [set.intersection]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  O
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  O
    set_intersection(Rng1&& rng1, Rng2&& rng2, O result,
                     Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

11.5.5.4 `set_difference` [set.difference]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(I1), tag::out(O)>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, O result,
                   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `{last1, result + ` n`}`, where n is the number of elements in the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m-n, 0)$ elements from `[first1, last1)` shall be copied to the output range.

11.5.5.5 `set_symmetric_difference` [set.symmetric.difference]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                             Comp comp = Comp{}, Proj1 proj1 = Proj1{},
                             Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O,
    class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
  requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, O, Comp, Proj1, Proj2>
  tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
               tag::in2(safe_iterator_t<Rng2>),
               tag::out(O)>
    set_symmetric_difference(Rng1&& rng1, Rng2&& rng2, O result, Comp comp = Comp{},
                             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* `make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result + ` n`)`, where n is the number of elements in the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` applications of the comparison function and projections.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m > n$, and the last $n-m$ of these elements from `[first2,last2)` if $m < n$.

11.5.6 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.

11.5.6.1 `push_heap` [push.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I push_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    push_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

2 *Requires:* The range `[first,last - 1)` shall be a valid heap.

3 *Returns:* `last`

4 *Complexity:* At most `log(last - first)` applications of the comparison function and projection.

11.5.6.2 `pop_heap` [pop.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    pop_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The range `[first,last)` shall be a valid non-empty heap.

2 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

3 *Returns:* `last`

4 *Complexity:* At most `2 * log(last - first)` applications of the comparison function and projection.

11.5.6.3 `make_heap` [make.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Constructs a heap out of the range `[first,last)`.

2 *Returns:* `last`

3 *Complexity:* At most `3 * (last - first)` applications of the comparison function and projection.

11.5.6.4 `sort_heap` [sort.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
```

```
  requires Sortable<I, Comp, Proj>
  I sort_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  safe_iterator_t<Rng>
    sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Sorts elements in the heap `[first,last)`.

2 *Requires:* The range `[first,last)` shall be a valid heap.

3 *Returns:* `last`

4 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`), and exactly twice as many applications of the projection.

11.5.6.5 `is_heap` [is.heap]

```
template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  bool is_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  bool
    is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* `is_heap_until(first, last, comp, proj) == last`

```
template <RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I is_heap_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <RandomAccessRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

2 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is a heap.

3 *Complexity:* Linear.

11.5.7 Minimum and maximum [alg.min.max]

```
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& min(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Returns:* The smaller value.

2 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T min(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
```

```
  value_type_t<iterator_t<Rng>>
    min(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

3 *Requires:* `distance(rng) > 0`.

4 *Returns:* The smallest value in the `initializer_list` or range.

5 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

```
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

6 *Returns:* The larger value.

7 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr T max(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  value_type_t<iterator_t<Rng>>
    max(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

8 *Requires:* `distance(rng) > 0`.

9 *Returns:* The largest value in the `initializer_list` or range.

10 *Remarks:* Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

```
template <class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(const T&), tag::max(const T&)>
    minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
```

11 *Returns:* `{b, a}` if `b` is smaller than `a`, and `{a, b}` otherwise.

12 *Remarks:* Returns `{a, b}` when the arguments are equivalent.

13 *Complexity:* Exactly one comparison and exactly two applications of the projection.

```
template <Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
  constexpr tagged_pair<tag::min(T), tag::max(T)>
    minmax(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj> Comp = less<>>
  requires Copyable<value_type_t<iterator_t<Rng>>>
  tagged_pair<tag::min(value_type_t<iterator_t<Rng>>),
              tag::max(value_type_t<iterator_t<Rng>>)>
    minmax(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

14 *Requires:* `distance(rng) > 0`.

15 *Returns:* `{x, y}`, where `x` has the smallest and `y` has the largest value in the `initializer_list` or range.

16 *Remarks:* x is a copy of the leftmost argument when several arguments are equivalent to the smallest. y is a copy of the rightmost argument when several arguments are equivalent to the largest.

17 *Complexity:* At most `(3/2) * distance(rng)` applications of the corresponding predicate, and at most twice as many applications of the projection.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

18 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false`. Returns `last` if `first == last`.

19 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  safe_iterator_t<Rng>
    max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

20 *Returns:* The first iterator `i` in the range `[first,last)` such that for every iterator `j` in the range `[first,last)` the following corresponding condition holds:
`invoke(comp, invoke(proj, *i), invoke(proj, *j)) == false`. Returns `last` if `first == last`.

21 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the comparison function and exactly twice as many applications of the projection.

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::min(I), tag::max(I)>
    minmax_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
  tagged_pair<tag::min(safe_iterator_t<Rng>),
              tag::max(safe_iterator_t<Rng>)>
    minmax_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

22 *Returns:* `{first, first}` if `[first,last)` is empty, otherwise `{m, M}`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator in `[first,last)` such that no iterator in the range refers to a larger element.

23 *Complexity:* At most $max(\lfloor\frac{3}{2}(N-1)\rfloor, 0)$ applications of the comparison function and at most twice as many applications of the projection, where N is `distance(first, last)`.

11.5.8 Lexicographical comparison [alg.lex.comparison]

```
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
  bool
    lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
                            Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
      projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
  bool
    lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
                            Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

1 *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)` and `false` otherwise.

2 *Complexity:* At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison and projections.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
  if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
}
return first1 == last1 && first2 != last2;
```

4 *Remarks:* An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

11.5.9 Permutation generators [alg.permutation.generators]

```
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>,
    class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  bool
    next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

2 *Complexity:* At most `(last - first)/2` swaps.

```
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
    class Proj = identity>
  requires Sortable<I, Comp, Proj>
  bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>,
    class Proj = identity>
  requires Sortable<iterator_t<Rng>, Comp, Proj>
  bool
    prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

3 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `comp` and `proj`.

4 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

5 *Complexity:* At most `(last - first)/2` swaps.

12 Numerics library [numerics]

Header `<experimental/ranges/random>` synopsis

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <class G>
  concept bool UniformRandomNumberGenerator = see below;
}}}}
```

12.1 Uniform random number generator requirements [rand.req.urng]

```
template <class G>
concept bool UniformRandomNumberGenerator =
  Invocable<G&> &&
  UnsignedIntegral<result_of_t<G&()>> &&
  requires {
    { G::min() } -> Same<result_of_t<G&()>>&&;
    { G::max() } -> Same<result_of_t<G&()>>&&;
  };
```

1 A *uniform random number generator* `g` of type `G` is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note:* The degree to which `g`'s results approximate the ideal is often determined statistically. — *end note*]

2 Let `g` be any object of type `G`. `UniformRandomNumberGenerator<G>` is satisfied only if

(2.1) — Both `G::min()` and `G::max()` are constant expressions (ISO/IEC 14882:2014 §5.19).

(2.2) — `G::min() < G::max()`.

(2.3) — `G::min() <= g()`.

(2.4) — `g() <= G::max()`.

(2.5) — `g()` has amortized constant complexity.

Annex A (normative) Compatibility features [depr]

1 This Clause describes features of this document that are specified for compatibility with existing implementations.

2 These are deprecated features, where *deprecated* is defined as: Normative for the current edition of the Ranges TS, but having been identified as a candidate for removal from future revisions. An implementation may declare library names and entities described in this section with the `deprecated` attribute (ISO/IEC 14882:2014 §7.6.5).

A.1 Rvalue range access [depr.rvalue.ranges]

1 Use of the range access customization point objects `begin`, `end`, `cbegin`, `cend`, `rbegin`, `rend`, `crbegin`, `crend`, `data`, and `cdata` with rvalue arguments is deprecated. In a future revision of this document, such usage could become ill-formed.

A.2 Range-and-a-half algorithms [depr.algo.range-and-a-half]

1 The following algorithm signatures are deemed unsafe and are deprecated in this document.

2 Overloads of algorithms that take a `Range` argument and a forwarding reference parameter `first2_` behave as if they are implemented by calling `begin` and `end` on the `Range` and dispatching to the overload that takes separate iterator and sentinel arguments, perfectly forwarding `first2_`.

```
template <InputIterator I1, Sentinel<I1> S1, class I2, class R = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires InputIterator<decay_t<I2>> && !Range<I2> &&
    IndirectRelation<R, projected<I1, Proj1>, projected<decay_t<I2>, Proj2>>
  tagged_pair<tag::in1(I1), tag::in2(decay_t<I2>)>
    mismatch(I1 first1, S1 last1, I2&& first2_, R comp = R{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, class I2, class R = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires InputIterator<decay_t<I2>> && !Range<I2> &&
    IndirectRelation<R, projected<iterator_t<Rng1>, Proj1>, projected<decay_t<I2>, Proj2>>
  tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(decay_t<I2>)>
    mismatch(Rng1&& rng1, I2&& first2_, R comp = R{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

3 *Effects:* Equivalent to:

```
return mismatch(first1, last1, std::forward<I2>(first2_), unreachable{}, comp, proj1, proj2);
```

except that the underlying algorithm never increments `first2` more than `last1 - first1` times.

```
template <InputIterator I1, Sentinel<I1> S1, class I2,
    class R = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires InputIterator<decay_t<I2>> && !Range<I2> &&
    IndirectlyComparable<I1, decay_t<I2>, R, Proj1, Proj2>
  bool equal(I1 first1, S1 last1, I2&& first2_, R comp = R{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
template <InputRange Rng1, class I2, class R = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires InputIterator<decay_t<I2>> && !Range<I2> &&
    IndirectlyComparable<iterator_t<Rng1>, decay_t<I2>, R, Proj1, Proj2>
  bool equal(Rng1&& rng1, I2&& first2_, R comp = R{},
             Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

4 *Effects:* Equivalent to:

```
return first1 == mismatch(first1, last1, std::forward<I2>(first2_), comp, proj1, proj2).in1();
```

```
template <ForwardIterator I1, Sentinel<I1> S1, class I2,
    class R = equal_to<>, class Proj1 = identity, class Proj2 = identity>
  requires ForwardIterator<decay_t<I2>> && !Range<I2> &&
    IndirectlyComparable<I1, decay_t<I2>, R, Proj1, Proj2>
  bool is_permutation(I1 first1, S1 last1, I2&& first2_, R comp = R{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, class I2, class R = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
  requires ForwardIterator<decay_t<I2>> && !Range<I2> &&
    IndirectlyComparable<iterator_t<Rng1>, decay_t<I2>, R, Proj1, Proj2>
  bool is_permutation(Rng1&& rng1, I2&& first2_, R comp = R{},
                      Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

5 *Effects:* Equivalent to:

```
  auto first2 = std::forward<I2>(first2_);
  return is_permutation(first1, last1, first2, next(first2, distance(first1, last1)),
                        comp, proj1, proj2);
```

```
template <ForwardIterator I1, Sentinel<I1> S1, class I2>
  requires ForwardIterator<decay_t<I2>> && !Range<I2> &&
    IndirectlySwappable<I1, decay_t<I2>>
  tagged_pair<tag::in1(I1), tag::in2(decay_t<I2>)>
    swap_ranges(I1 first1, S1 last1, I2&& first2_);

template <ForwardRange Rng, class I2>
  requires ForwardIterator<decay_t<I2>> && !Range<I2> &&
    IndirectlySwappable<iterator_t<Rng>, decay_t<I2>>
  tagged_pair<tag::in1(safe_iterator_t<Rng>), tag::in2(decay_t<I2>)>
    swap_ranges(Rng&& rng1, I2&& first2_);
```

6 *Effects:* Equivalent to:

```
  auto first2 = std::forward<I2>(first2_);
  return swap_ranges(first1, last1, first2, next(first2, distance(first1, last1)));
```

```
template <InputIterator I1, Sentinel<I1> S1, class I2, WeaklyIncrementable O,
    CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
  requires InputIterator<decay_t<I2>> && !Range<I2> &&
    Writable<O, indirect_result_of_t<F&(projected<I1, Proj1>,
        projected<decay_t<I2>, Proj2>)>>
  tagged_tuple<tag::in1(I1), tag::in2(decay_t<I2>), tag::out(O)>
    transform(I1 first1, S1 last1, I2&& first2_, O result,
              F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

```
template <InputRange Rng, class I2, WeaklyIncrementable O, CopyConstructible F,
    class Proj1 = identity, class Proj2 = identity>
  requires InputIterator<decay_t<I2>> && !Range<I2> &&
    Writable<O, indirect_result_of_t<F&(
        projected<iterator_t<Rng>, Proj1>, projected<decay_t<I2>, Proj2>>)>
  tagged_tuple<tag::in1(safe_iterator_t<Rng>), tag::in2(decay_t<I2>), tag::out(O)>
    transform(Rng&& rng1, I2&& first2_, O result,
              F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

7 *Effects:* Equivalent to:

```
return transform(first1, last1, std::forward<I2>(first2_), unreachable{}, pred, proj1, proj2);
```

except that the underlying algorithm never increments `first2` more than `last1 - first1` times.

Annex B (informative) Acknowledgements [acknowledgements]

The design of this specification is based, in part, on a concept specification of the algorithms part of the C++ standard library, known as the "Palo Alto report" ([5]), which was developed by a large group of experts as a test of the expressive power of the idea of concepts.

The authors would like to thank Andrew Sutton for his work on the Concepts TS (2), for his help formalizing the ideas of the range-v3 library [2] on which this work is based, and for his review of this document.

Sean Parent has made major contributions to both the foundations and the wording of this document.

Stephan T. Lavavej offered a careful review of much of this document, a non-trivial undertaking.

We would also like to thank the members of the Ranges SG who offered feedback on early drafts, especially Tony Van Eerd and Walter Brown.

Christopher Di Bella has contributed many editorial fixes.

This work was made possible by a generous grant from the Standard C++ Foundation.

Annex C (informative) Compatibility [diff]

C.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.

C.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 ([3]) for a discussion of the issues.

C.1.2 Stronger Constraints [diff.cpp.constraints]

1 In this proposal, many algorithms and utilities get stricter type checking. For example, algorithms constrained with `LessThanComparable` today are constrained by `StrictTotallyOrdered` in this document. This concept requires types to provide *all* the relational operators, not just `operator<`.

2 The use of coarser-grained, higher-level concepts in algorithm constraints is to make the type checks more semantic in nature and less syntactic. It also has the benefit of being less verbose while giving algorithm implementors greater implementation freedom. This approach is in contrast to the previous effort to add concepts to the Standard Library in the C++0x timeframe, which saw a proliferation of small, purely syntactic concepts and algorithm constraints that merely restated the algorithms' implementation details more verbosely in the algorithms' function signatures.

3 The potential for breakage must be carefully weighed against the integrity and complexity of the constraints system. The coarseness of the concepts may need to change in response to real-world usage.

C.1.3 Constrained Functional Objects [diff.cpp.functional]

1 The algorithm design described in this document assumes that the function objects `std::equal_to` and `std::less` get constraints added to their function call operators. (The former is constrained with `EqualityComparable` and the latter with `StrictTotallyOrdered`). Similar constraints are added to the other function objects in `<functional>`. As with the coarsely-grained algorithm constraints, these function object constraints are likely to cause user code to break.

2 Real-world experience is needed to assess the seriousness of the breakage. From a correctness point of view, the constraints are logical and valuable, but it's possible that for the sake of compatibility we provide both constrained and unconstrained functional objects.

C.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 something like `std::optional`, as specified in the C++ 17 Working Draft N4618 §20.6). 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.

C.1.5 iterator_traits cannot be specialized [diff.cpp.iteratortraits]

1 In this STL design, `iterator_traits` changes from being a class template to being an alias template. This is to intentionally break any code that tries to specialize it. In its place are the three class templates `difference_type`, `value_type`, and `iterator_category`. The need for this traits balkanization is because the associated types belong to separate concepts: `difference_type` belongs to `WeaklyIncrementable`; `value_type` belongs to `Readable`; and `iterator_category` belongs to `InputIterator`.

2 This breakage is intentional and inherent in the decomposition of the iterator concepts established by the Palo Alto report ([5]).

C.2 Ranges and the Palo Alto TR (N3351) [diff.n3351]

1 The Palo Alto report ([5]) 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.

C.2.1 Sentinels [diff.n3351.sentinels]

1 In the design presented in this document, the type of a range's end delimiter may differ from the iterator representing the range's start position. The reasons for this change are described in N4128 ([3]). This causes a number of differences from the Palo Alto report:

(1.1) — The algorithms get an additional constraint for the sentinel.

(1.2) — The return types of the algorithms are changed as described above (C.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.

C.2.2 Invocables and Projections [diff.n3351.invok_proj]

1 Adobe's Source Libraries [1] pioneered the use of *callables* and *projections* in the standard algorithms. Invocables 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.

C.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 document feels this is potentially a rich source of confusion. This document hews closer to the current standard by having only one associated type, `DifferenceType`, and requiring it to be signed.

C.2.4 Distance Primitive is $\mathcal{O}(1)$ 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 $\mathcal{O}(N)$. It reads:

> The standard mandates a different definition for random access iterators: `distance(i, j) == j - i`. We see this as a specification error; the guarantees of the `distance` operation have been weakened for an iterator specialization.
>
> In our design, we consider the two operations to be distinct.

The design presented in this document keeps the specialization for random access iterators. To do otherwise would be to silently break complexity guarantees in an unknown amount of working code.

To address the concern about weakened guarantees of the `distance` primitive, the design presented here requires that random access iterators model `SizedSentinel` (9.3.10). The `SizedSentinel` concept requires that `b - a` return the number of times `a` would have to be incremented to make it compare equal to `b`. Any type purporting to be a random access iterator that fails to meet that requirement is by definition not a valid random access iterator.

C.2.5 Output Iterators [diff.n3351.output_iters]

1 The Palo Alto report does not define concepts for output iterators, making do with `WeaklyIncrementable`, `Writable`, and (where needed) `EqualityComparable`. The author of this document sees little downside to grouping these into the familiar `OutputIterator` concept. Even if not strictly needed, its absence would be surprising.

C.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 algorithms as described in The Palo Alto report feels like it would be best done in a separate proposal.

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, Sean Parent, and Andrew Sutton. N4128: Ranges for the standard library, revision 1, 10 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html.

[4] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[5] Bjarne Stroustrup and Andrew Sutton. N3351: A concept design for the stl, 1 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf.

Index

argument, 10

C++ Standard, 2
class
    base, 10
Concepts TS, 2
constant iterator, 52
constant subexpression, 3
conventions, 9

expression-equivalent, 3

function
    virtual member, 10

library
    Ranges, 7, 10

multi-pass guarantee, 61
mutable iterator, 52

namespace, 9

projection, 3

requirements, 8
    iterator, 52
    uniform random number generator, 170
restriction, 10

statement
    iteration, 6
swappable, 19
swappable with, 19

undefined behavior, 105
uniform random number generator
    requirements, 170
unspecified, 156

Index of library names

adjacent_find, 141
advance, 69
all_of, 138
any_of, 138
Assignable, 18

back_insert_iterator, 77
  back_insert_iterator, 77
  constructor, 77
back_inserter, 78
base
  counted_iterator, 95
  move_iterator, 83
  reverse_iterator, 73
bidirectional_iterator_tag, 68, 69
BidirectionalIterator, 61
binary_search, 159
Boolean, 21

Common, 17
common_iterator, 88
  common_iterator, 89
  constructor, 89
  iter_move, 92
  iter_swap, 92
  operator!=, 91
  operator*, 90
  operator++, 91
  operator-, 91
  operator->, 90
  operator=, 90
  operator==, 91
CommonReference, 17
Constructible, 20
ConvertibleTo, 16
copy, 144
copy_backward, 145
copy_n, 144, 145
Copyable, 24
CopyConstructible, 20
count, 141
  counted_iterator, 95
count_if, 141
counted_iterator, 93
  base, 95
  constructor, 95
  count, 95
  counted_iterator, 95
  iter_move, 99
  iter_swap, 99
  operator!=, 97
  operator*, 95
  operator+, 96, 98
  operator++, 95, 96
  operator+=, 96
  operator-, 97, 98
  operator-=, 97
  operator--, 96
  operator<, 97
  operator<=, 98
  operator=, 95
  operator==, 97
  operator>, 98
  operator>=, 98
  operator[], 97

dangling, 99
  dangling, 100
  get_unsafe, 100
default_sentinel, 92
DefaultConstructible, 20
DerivedFrom, 16
Destructible, 20
difference_type, 54
difference_type_t, 54
distance, 70

empty, 69
equal, 142
  istreambuf_iterator, 107
equal_range, 159
equal_to, 30
equal_to<>, 31
EqualityComparable, 22
EqualityComparableWith, 22
<experimental/ranges/algorithm>, 119
<experimental/ranges/concepts>, 13
<experimental/ranges/iterator>, 43
<experimental/ranges/range>, 110
<experimental/ranges/utility>, 27

failed

  ostreambuf_iterator, 109
fill, 149
fill_n, 149
find, 139
find_end, 140
find_first_of, 140
find_if, 139
find_if_not, 139
for_each, 139
forward_iterator_tag, 68, 69
ForwardIterator, 60
front_insert_iterator, 78
  constructor, 78
  front_insert_iterator, 78
front_inserter, 79

generate, 149
generate_n, 149
get_unsafe
  dangling, 100
greater, 30
greater<>, 31
greater_equal, 30
greater_equal<>, 32

identity, 32
includes, 161
Incrementable, 58
indirect_result_of, 62
IndirectlyComparable, 65
IndirectlyCopyable, 64
IndirectlyCopyableStorable, 64
IndirectlyMovable, 64
IndirectlyMovableStorable, 64
IndirectlySwappable, 65
IndirectRegularUnaryInvocable, 62
IndirectRelation, 62
IndirectStrictWeakOrder, 62
IndirectUnaryInvocable, 62
IndirectUnaryPredicate, 62
inplace_merge, 160
input_iterator_tag, 68, 69
InputIterator, 60
insert_iterator, 79
  constructor, 80
  insert_iterator, 79
inserter, 80
Integral, 17
Invocable, 24
is_heap, 165
is_heap_until, 165
is_nothrow_swappable, 35
is_nothrow_swappable_with, 34
is_partitioned, 153
is_permutation, 143
is_sorted, 157
is_sorted_until, 157
is_swappable, 34
is_swappable_with, 34
istream_iterator, 101
  constructor, 102
  destructor, 102
  operator!=, 103
  operator*, 103
  operator++, 103
  operator->, 103
  operator==, 103
istreambuf_iterator, 105
  constructor, 106, 107
  operator++, 107
iter_move
  common_iterator, 92
  counted_iterator, 99
  move_iterator, 85
  reverse_iterator, 76
iter_swap
  common_iterator, 92
  counted_iterator, 99
  move_iterator, 85
  reverse_iterator, 76
Iterator, 58
iterator_category, 56
iterator_category_t, 56
iterator_traits, 66

less, 30
less<>, 31
less_equal, 30
less_equal<>, 32
lexicographical_compare, 168
lower_bound, 158

make_counted_iterator, 99
make_heap, 164
make_move_iterator, 85
make_move_sentinel, 87
make_reverse_iterator, 76
make_tagged_pair, 41
make_tagged_tuple, 42
max, 166
max_element, 167
merge, 160

Mergeable, 65
min, 165
min_element, 167
minmax, 166
minmax_element, 167
mismatch, 142
Movable, 24
movemove, 145
move_backward, 146
move_iterator, 81
    base, 83
    constructor, 82
    iter_move, 85
    iter_swap, 85
    move_iterator, 82
    operator!=, 84
    operator*, 83
    operator+, 83, 85
    operator++, 83
    operator+=, 84
    operator-, 84, 85
    operator-=, 84
    operator--, 83
    operator<, 84
    operator<=, 84
    operator=, 83
    operator==, 84
    operator>, 84
    operator>=, 85
    operator[], 84
move_sentinel, 86
    constructor, 87
    move_sentinel, 87
    operator!=, 87
    operator-, 87
    operator=, 87
    operator==, 87
MoveConstructible, 20

next, 70
next_permutation, 168
none_of, 139
not_equal_to, 30
not_equal_to<>, 31
nth_element, 157

operator!=
    common_iterator, 91
    counted_iterator, 97
    istream_iterator, 103
    istreambuf_iterator, 107
    move_iterator, 84
    move_sentinel, 87
    reverse_iterator, 75
    unreachable, 101
operator*
    back_insert_iterator, 77
    common_iterator, 90
    counted_iterator, 95
    front_insert_iterator, 79
    insert_iterator, 80
    istream_iterator, 103
    istreambuf_iterator, 107
    move_iterator, 83
    ostream_iterator, 105
    ostreambuf_iterator, 109
    reverse_iterator, 73
operator+
    counted_iterator, 96, 98
    move_iterator, 83, 85
    reverse_iterator, 74, 76
operator++
    back_insert_iterator, 78
    common_iterator, 91
    counted_iterator, 95, 96
    front_insert_iterator, 79
    insert_iterator, 80
    istream_iterator, 103
    istreambuf_iterator, 107
    move_iterator, 83
    ostream_iterator, 105
    ostreambuf_iterator, 109
    reverse_iterator, 73
operator+=
    counted_iterator, 96
    move_iterator, 84
    reverse_iterator, 74
operator-
    common_iterator, 91
    counted_iterator, 97, 98
    move_iterator, 84, 85
    move_sentinel, 87
    reverse_iterator, 74, 75
operator-=
    counted_iterator, 97
    move_iterator, 84
    reverse_iterator, 74
operator->
    common_iterator, 90
    istream_iterator, 103
    reverse_iterator, 73
operator--

counted_iterator, 96
move_iterator, 83
reverse_iterator, 73, 74
operator<
counted_iterator, 97
move_iterator, 84
reverse_iterator, 75
operator<=
counted_iterator, 98
move_iterator, 84
reverse_iterator, 75
operator=
reverse_iterator, 73
back_insert_iterator, 77
common_iterator, 90
counted_iterator, 95
front_insert_iterator, 78, 79
insert_iterator, 80
move_iterator, 83
move_sentinel, 87
ostream_iterator, 105
ostreambuf_iterator, 108
tagged, 39, 40
operator==
common_iterator, 91
counted_iterator, 97
istream_iterator, 103
istreambuf_iterator, 107
move_iterator, 84
move_sentinel, 87
reverse_iterator, 74
unreachable, 100
operator>
counted_iterator, 98
move_iterator, 84
reverse_iterator, 75
operator>=
counted_iterator, 98
move_iterator, 85
reverse_iterator, 75
operator[]
counted_iterator, 97
move_iterator, 84
reverse_iterator, 74
ostream_iterator, 104
constructor, 104
destructor, 104
operator*, 105
operator++, 105
operator=, 105
ostreambuf_iterator, 108
constructor, 108
output_iterator_tag, 68, 69
OutputIterator, 60

partial_sort, 156
partial_sort_copy, 157
partition, 154
partition_copy, 154
partition_point, 155
Permutable, 65
pop_heap, 164
Predicate, 25
prev, 70
prev_permutation, 169
projected, 63
proxy
istreambuf_iterator, 106
push_heap, 164

random_access_iterator_tag, 68, 69
RandomAccessIterator, 61
Readable, 57
Regular, 24
RegularInvocable, 25
Relation, 25
remove, 149
remove_copy, 150
remove_copy_if, 150
remove_if, 149
replace, 147
replace_copy, 148
replace_copy_if, 148
replace_if, 147
reverse, 152
reverse_copy, 152
reverse_iterator, 71
reverse_iterator, 72
base, 73
constructor, 73
iter_move, 76
iter_swap, 76
make_reverse_iterator non-member function, 76
operator++, 73
operator--, 74
rotate, 152
rotate_copy, 153
rvalue_reference_t, 56

Same, 16
search, 143

search_n, 144
Semiregular, 24
Sentinel, 59
set_difference, 162
set_intersection, 162
set_symmetric_difference, 163
set_union, 161
shuffle, 153
SignedIntegral, 18
SizedSentinel, 59
sort, 155
sort_heap, 164
Sortable, 66
stable_partition, 154
stable_sort, 156
StrictTotallyOrdered, 22
swap, 28
    tagged, 40
    tagged, 40
swap_ranges, 146
Swappable, 18

tagged, 37
    operator=, 39, 40
    swap, 40
    tagged, 39
tagged_tuple
    make_tagged_tuple, 42
transform, 146
tuple_element, 40
tuple_size, 40

unique, 151
unique_copy, 151
unreachable, 100
    operator!=, 101
    operator==, 100
UnsignedIntegral, 18
upper_bound, 158

value_type, 55

WeaklyEqualityComparableWith, 22
WeaklyIncrementable, 57
Writable, 57