Programming Languages — C++ Extensions for Ranges
Langages de programmation — Extensions C++ pour les «ranges»

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document stage: (30) Committee
Document Language: E
# Contents

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
</tbody>
</table>

## 1 General

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2 References</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Implementation compliance</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Namespaces, headers, and modifications to standard classes</td>
<td>2</td>
</tr>
</tbody>
</table>

## 2 Statements

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Iteration statements</td>
<td>4</td>
</tr>
</tbody>
</table>

## 3 Library introduction

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 General</td>
<td>6</td>
</tr>
<tr>
<td>3.2 Definitions</td>
<td>6</td>
</tr>
<tr>
<td>3.3 Method of description (Informative)</td>
<td>6</td>
</tr>
<tr>
<td>3.4 Library-wide requirements</td>
<td>8</td>
</tr>
</tbody>
</table>

## 4 Concepts library

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 General</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Core language concepts</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Comparison concepts</td>
<td>16</td>
</tr>
<tr>
<td>4.4 Object concepts</td>
<td>19</td>
</tr>
<tr>
<td>4.5 Callable concepts</td>
<td>21</td>
</tr>
</tbody>
</table>

## 5 General utilities library

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 General</td>
<td>24</td>
</tr>
<tr>
<td>5.2 Utility components</td>
<td>24</td>
</tr>
<tr>
<td>5.3 Function objects</td>
<td>26</td>
</tr>
<tr>
<td>5.4 Metaprogramming and type traits</td>
<td>29</td>
</tr>
<tr>
<td>5.5 Tagged tuple-like types</td>
<td>34</td>
</tr>
</tbody>
</table>

## 6 Iterators library

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 General</td>
<td>40</td>
</tr>
<tr>
<td>6.2 Iterator requirements</td>
<td>40</td>
</tr>
<tr>
<td>6.3 Indirect callable requirements</td>
<td>48</td>
</tr>
<tr>
<td>6.4 Common algorithm requirements</td>
<td>50</td>
</tr>
<tr>
<td>6.5 Header <code>&lt;experimental/ranges/iterator&gt;</code> synopsis</td>
<td>52</td>
</tr>
<tr>
<td>6.6 Iterator primitives</td>
<td>60</td>
</tr>
<tr>
<td>6.7 Iterator adaptors</td>
<td>68</td>
</tr>
<tr>
<td>6.8 Stream iterators</td>
<td>96</td>
</tr>
<tr>
<td>6.9 Range concepts</td>
<td>104</td>
</tr>
<tr>
<td>6.10 Range access</td>
<td>108</td>
</tr>
<tr>
<td>6.11 Range primitives</td>
<td>110</td>
</tr>
</tbody>
</table>

## 7 Algorithms library

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>iii</td>
</tr>
</tbody>
</table>
Contents

7.1 General ......................................................... 113
7.2 Tag specifiers ................................................. 133
7.3 Non-modifying sequence operations ......................... 134
7.4 Mutating sequence operations ................................ 140
7.5 Sorting and related operations .............................. 152
7.6 C library algorithms ........................................... 165

8 Numerics library ................................................. 167
8.1 Uniform random number generator requirements .......... 167

A Compatibility features ........................................... 168
A.1 Rvalue range access ...................................... 168
A.2 Range-and-a-half algorithms .............................. 168

B Acknowledgements .............................................. 170

C Compatibility ................................................... 171
C.1 C++ and Ranges ........................................... 171
C.2 Ranges and the Palo Alto TR (N3351) ...................... 172

Bibliography ......................................................... 174

Index .............................................................. 175

Index of library names ........................................... 176
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ranges TS library headers</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Library categories</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Fundamental concepts library summary</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>General utilities library summary</td>
<td>24</td>
</tr>
<tr>
<td>49</td>
<td>Type property predicates</td>
<td>30</td>
</tr>
<tr>
<td>57</td>
<td>Other transformations</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Iterators library summary</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Relations among iterator categories</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Ranges library summary</td>
<td>104</td>
</tr>
<tr>
<td>8</td>
<td>Relations among range categories</td>
<td>105</td>
</tr>
<tr>
<td>9</td>
<td>Algorithms library summary</td>
<td>113</td>
</tr>
<tr>
<td>10</td>
<td>Header <code>&lt;stdlib&gt;</code> synopsis</td>
<td>165</td>
</tr>
</tbody>
</table>
1 General

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

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

—Author Unknown

1.1 Scope

This Technical Specification describes extensions to the C++ Programming Language (1.2) that permit operations on ranges of data. These extensions include changes and additions to the existing library facilities as well as the extension of 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 (1.2).

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 (6.10).

3 This paper 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 paper does not specify any new range views, actions, or facade or adaptor utilities; all are left as future work.
1.2 References

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

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

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

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

(1.4) JTC1/SC22/WG21 N3351, A Concept Design for the STL

ISO/IEC 14882:2014 is herein called the C++ Standard, N3351 is called the “The Palo Alto” report, and ISO/IEC TS 19217:2015 is called the Concepts TS.

1.3 Implementation compliance

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

1.4 Namespaces, headers, and modifications to standard classes

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

(1.1) 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 paper. 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 paper assumes that the contents of the std::experimental::ranges::v1 namespace will become a new constrained version of the C++ Standard Library that will be delivered alongside the existing unconstrained version.

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

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

#include <algorithm>
Table 1 — Ranges TS library headers

<table>
<thead>
<tr>
<th>&lt;experimental/ranges/algorithm&gt;</th>
<th>&lt;experimental/ranges/random&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;experimental/ranges/concepts&gt;</td>
<td>&lt;experimental/ranges/tuple&gt;</td>
</tr>
<tr>
<td>&lt;experimental/ranges/functional&gt;</td>
<td>&lt;experimental/ranges/utility&gt;</td>
</tr>
<tr>
<td>&lt;experimental/ranges/iterator&gt;</td>
<td></td>
</tr>
</tbody>
</table>
2 Statements

2.1 Iteration statements

2.1.1 The range-based for statement

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

For a range-based for statement of the form

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

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

```plaintext
(expression)
```

and for a range-based for statement of the form

```plaintext
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

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

and

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

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

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

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

3. Otherwise, `begin-expr` and `end-expr` are `begin(_range)` and `end(_range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (3.4.2). [Note: Ordinary unqualified lookup (3.4.1) is not performed. — end note]
In the decl-specifier-seq of a for-range-declaration, each declspecifier shall be either a type-specifier or constexpr. The decl-specifier-seq shall not define a class or enumeration.
3 Library introduction

3.1 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 3.4, Clauses 4 through 8, 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 4–8, as shown in Table 2.

Table 2 — Library categories

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Concepts library</td>
</tr>
<tr>
<td>5</td>
<td>General utilities library</td>
</tr>
<tr>
<td>6</td>
<td>Iterators library</td>
</tr>
<tr>
<td>7</td>
<td>Algorithms library</td>
</tr>
<tr>
<td>8</td>
<td>Numerics library</td>
</tr>
</tbody>
</table>

4 The concepts library (Clause 4) 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 5) includes components used by other library elements and components used as infrastructure in C++ programs, such as function objects.

6 The iterators library (Clause 6) 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 (6.9).

7 The algorithms library (Clause 7) describes components that C++ programs may use to perform algorithmic operations on containers (Clause ISO/IEC 14882:2014 §23) and other sequences.

8 The numerics library (Clause 8) provides concepts that are useful to constrain numeric algorithms.

3.2 Definitions

1 Terms defined in ISO/IEC 14882:2014 §17.3 are used in this document with the same meaning.

3.3 Method of description (Informative)

1 This subclause describes the conventions used to specify the Ranges library. 3.3.1 describes the structure of the normative Clauses 4 through 8 and Annex A. 3.3.2 describes other editorial conventions.

3.3.1 Structure of each clause

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

(1.1) — Summary
(1.2) — Requirements

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.
3.3.1.2 Summary

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.

1 Paragraphs labeled “Note(s):” or “Example(s):” are informative, other paragraphs are normative.

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

- macros
- values
- types
- classes and class templates
- functions and function templates
- objects
- concepts

3.3.1.3 Requirements

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 Template arguments
2 Derived classes
3 Containers, iterators, and algorithms that meet an interface convention or satisfy a concept

1 Interface convention requirements are stated as generally as possible. Instead of stating “class X has to define a member function \texttt{operator++(0)},” the interface requires “for any object \texttt{x} of class \texttt{X}, ++\texttt{x} is defined.” That is, whether the operator is a member is unspecified.

2 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 7) that uses the well-defined expression requirements is described in terms of the valid expressions for its formal type parameters.

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

4 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.\(^2\)

5 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. \([Example: The \texttt{operator<} of the \texttt{StrictTotallyOrdered} concept (4.3.4) does not meet the semantic requirements of that concept when operating on NaNs. — end example}\) This does not affect whether a type satisfies the concept.

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

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

3.3.1.4 Detailed specifications

The detailed specifications of each entity defined in Clauses 4–8 follows the conventions established by ISO/IEC 14882:2014 §17.5.1.4.

3.3.2 Other conventions

This subclause describes several editorial conventions used to describe the contents of the Ranges library. These conventions are for describing member functions (3.3.2.1), private members (3.3.2.2), and customization point objects (3.3.2.3).

3.3.2.1 Functions within classes

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

3.3.2.2 Private members

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

3.3.2.3 Customization Point Objects

A customization point object is a function object (5.3) 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 (4.4.8).

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

4 The type of a customization point object T shall satisfy Invocable<const T, Args...>() (4.5.2) when the types of Args... meet the requirements specified in that customization point object’s definition. Otherwise, T shall not have a function call operator that participates in overload resolution.

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

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

[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]

3.4 Library-wide requirements

This subclause specifies requirements that apply to the entire Ranges library. Clauses 4 through 8 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.
3.4.1 **Library contents and organization**

3.4.1.1 describes the entities defined in the Ranges library.

3.4.2 **Using the library**

3.4.2.1 **Overview**

This section describes how a C++ program gains access to the facilities of the Ranges library. 3.4.2.2 describes effects during translation phase 4, while 3.4.2.3 describes effects during phase 8 (ISO/IEC 14882:2014 §2.2).

3.4.2.2 **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.

3.4.2.3 **Linkage**

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).

3.4.3 **Constraints on programs**

3.4.3.1 **Overview**

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 (3.4.3.2) and other constraints.

3.4.3.2 **Derived classes**

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).

3.4.3.3 **Other functions**

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.

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.

3.4.3.4 **Function arguments**

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.
3.4.3.5 Library object access

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.

3.4.3.6 Requires paragraph

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.

3.4.4 Conforming implementations

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.
4 Concepts library

4.1 General

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.

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

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 Core language concepts</td>
<td>&lt;experimental/ranges/concepts&gt;</td>
</tr>
<tr>
<td>4.3 Comparison concepts</td>
<td></td>
</tr>
<tr>
<td>4.4 Object concepts</td>
<td></td>
</tr>
<tr>
<td>4.5 Callable concepts</td>
<td></td>
</tr>
</tbody>
</table>

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

4.1.1 Equality Preservation

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.

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.

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)

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

An expression that may alter the value of one or more of its inputs in a manner observable to equality preserving expressions is said to modify those inputs. This document uses a notational convention to specify which expressions declared in a `requires-expression` modify which inputs: except where otherwise specified, an expression operand that is a non-constant lvalue or rvalue for the given operand is also required except where such an expression variant is explicitly required with

§ 4.1.1
differing semantics. Such implicit expression variants must meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of these implicit expression variants is unspecified.

[Example:

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

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

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

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

had been declared as well.

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

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

```cpp
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]

4.2 Core language concepts [concepts.lib.corelang]

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

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

4.2.2 Concept Same [concepts.lib.corelang.same]

```cpp
template <class T, class U>
concept bool Same() {
  return see below;
}
```
Same<T, U>() is satisfied if and only if T and U denote the same type.

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

4.2.3 Concept DerivedFrom

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

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

4.2.4 ConceptConvertibleTo

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

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

4.2.5 Concept CommonReference

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]

```cpp
template <class T, class U>
concept bool CommonReference() {
    return requires(T (&t)(), U (&u)()) {
        typename common_reference_t<T, U>;
        typename common_reference_t<U, T>;
        requires Same<common_reference_t<T, U>, common_reference_t<U, T>>();
        common_reference_t<T, U>(t());
        common_reference_t<T, U>(u());
    };
}
```

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 if and only if:

1. C(t()) equals C(t()) if and only if t() is an equality preserving expression (4.1.1).
2. C(u()) equals C(u()) if and only if u() is an equality preserving expression.

[Note: Users can customize the behavior of CommonReference by specializing the basic_common_reference class template (5.4.3). — end note]

4.2.6 Concept Common

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]

```cpp
template <class T, class U>
concept bool Common() {
    return CommonReference<const T&, const U&>();
}
```


typename common_type_t<T, U>;

typename common_type_t<U, T>;

requires Same<common_type_t<U, T>, common_type_t<T, U>>();

common_type_t<T, U>(t());

common_type_t<T, U>(u());

requires CommonReference<add_lvalue_reference_t<common_type_t<T, U>>,
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 if and only if:

(2.1) \[ C(t()) \text{ equals } C(t()) \text{ if and only if } t() \text{ is an equality preserving expression (4.1.1).} \]

(2.2) \[ C(u()) \text{ equals } C(u()) \text{ if and only if } u() \text{ is an equality preserving expression (4.1.1).} \]

3

[Note: Users can customize the behavior of Common by specializing the common_type class template (5.4.2). —end note]

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

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

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

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

1

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

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

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

1

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

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

template <class T, class U>
concept bool Assignable() {
    return CommonReference<const T&, const U&>() && requires(T&& t, U& u) {
        { std::forward<T>(t) = std::forward<U>(u) } -> Same<T&>;
    };
}

1

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

§ 4.2.10
--- std::addressof(t = v) == std::addressof(t).
--- After evaluating t = v:
--- t is equal to uu.
--- If v is a non-const rvalue, its resulting state is unspecified. [Note: v must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified. —end note]
--- Otherwise, v is not modified.

4.2.11 Concept Swappable

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

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

1 This subclause provides definitions for swappable types and expressions. In these definitions, let t denote an expression of type T, and let u denote an expression of type U.
2 An object t is swappable with an object u if and only if Swappable< T, U>() is satisfied. Swappable< T, U>() is satisfied if and only if given distinct objects tt equal to t and uu equal to u, after evaluating either ranges::swap(t, u) or ranges::swap(u, t), tt is equal to u and uu is equal to t.
3 An rvalue or lvalue t is swappable if and only if t is swappable with any rvalue or lvalue, respectively, of type T.

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

```cpp
#include <utility>

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

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

4.3 Comparison concepts

4.3.1 In general

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

4.3.2 Concept Boolean

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

template <class B>
concept bool Boolean() {
    return MoveConstructible<B>() && (see 4.4.4)
    requires(const B b1, const B b2, const bool a) {
        bool(b1);
        { b1 } -> bool;
        bool(!b1);
        { !b1 } -> bool;
        { b1 && b2 } -> Same<bool>;
        { b1 && a } -> Same<bool>;
        { a && b1 } -> Same<bool>;
        { b1 || b2 } -> Same<bool>;
        { b1 || a } -> Same<bool>;
        { a || b1 } -> Same<bool>;
        { b1 == b2 } -> bool;
        { b1 != b2 } -> bool;
        { b1 == a } -> bool;
        { a == b1 } -> bool;
        { b1 != a } -> bool;
        { a != b1 } -> bool;
    }

§ 4.3.2
Given values \( b_1 \) and \( b_2 \) of type \( B \), then \( \text{Boolean}<\text{B}>() \) is satisfied if and only if

(2.1) \( \text{bool}(b_1) = \begin{cases} \lambda (\text{bool} x) \{ \text{return } x; \} (b_1). \end{cases} \)

(2.2) \( \text{bool}(b_1) = !\text{bool}(!b_1). \)

(2.3) \( (b_1 \& \& b_2), (b_1 \& \& \text{bool}(b_2)), \) and \( \text{bool}(b_1) \& \& b_2 \) are all equal to \( \text{bool}(b_1) \& \& \text{bool}(b_2) \), and have the same short-circuit evaluation.

(2.4) \( (b_1 || b_2), (b_1 || \text{bool}(b_2)), \) and \( \text{bool}(b_1) || b_2 \) are all equal to \( \text{bool}(b_1) || \text{bool}(b_2) \), and have the same short-circuit evaluation.

(2.5) \( \text{bool}(b_1 == b_2), \text{bool}(b_1 == \text{bool}(b_2)), \) and \( \text{bool}(\text{bool}(b_1) == b_2) \) are all equal to \( \text{bool}(b_1 == \text{bool}(b_2)) \).

(2.6) \( \text{bool}(b_1 != b_2), \text{bool}(b_1 != \text{bool}(b_2)), \) and \( \text{bool}(\text{bool}(b_1) != b_2) \) are all equal to \( \text{bool}(b_1 != \text{bool}(b_2)) \).

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

4.3.3 Concept EqualityComparable

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

Let \( t \) and \( u \) be objects of types \( T \) and \( U \). \( \text{WeaklyEqualityComparable}<T, U>() \) is satisfied if and only if:

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

(1.2) \( \text{bool}(u == t) == \text{bool}(t == u). \)

(1.3) \( \text{bool}(t != u) == !\text{bool}(t == u). \)

(1.4) \( \text{bool}(u != t) == \text{bool}(t != u). \)

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

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

(2.1) \( \text{bool}(a == b) \) if and only if \( a \) is equal to \( b. \)

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

§ 4.3.3
template <class T, class U>
concept bool EqualityComparable() {
    return CommonReference<const T&, const U&>() &&
         EqualityComparable<T>() &&
         EqualityComparable<U>() &&
         EqualityComparable<
            remove_cv_t<remove_reference_t<common_reference_t<const T&, const U&>>>() &&
         WeaklyEqualityComparable<T, U>();
}

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

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

4.3.4 Concept StrictTotallyOrdered  [concepts.lib.compare.stricttotallyordered]

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

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

(1.1) Exactly one of bool(a < b), bool(b < a), or bool(a == b) is true.
(1.2) If bool(a < b) and bool(b < c), then bool(a < c).
(1.3) bool(a > b) == bool(b < a).
(1.4) bool(a <= b) == !bool(b < a).
(1.5) bool(a >= b) == !bool(a < b).

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

\$ 4.3.4 \$
Let \( t \) be an object of type \( T \), \( u \) be an object of type \( U \), and \( C \) be \( \text{common\_reference\_t<}\text{const } T&,\text{ const } U&>\). Then \( \text{StrictTotallyOrdered}<T, U>() \) is satisfied if and only if

\[
\begin{align*}
(2.1) &\quad \text{bool}(t < u) == \text{bool}(C(t) < C(u)). \\
(2.2) &\quad \text{bool}(t > u) == \text{bool}(C(t) > C(u)). \\
(2.3) &\quad \text{bool}(t \leq u) == \text{bool}(C(t) \leq C(u)). \\
(2.4) &\quad \text{bool}(t \geq u) == \text{bool}(C(t) \geq C(u)). \\
(2.5) &\quad \text{bool}(u < t) == \text{bool}(C(u) < C(t)). \\
(2.6) &\quad \text{bool}(u > t) == \text{bool}(C(u) > C(t)). \\
(2.7) &\quad \text{bool}(u \leq t) == \text{bool}(C(u) \leq C(t)). \\
(2.8) &\quad \text{bool}(u \geq t) == \text{bool}(C(u) \geq C(t)).
\end{align*}
\]

### 4.4 Object concepts

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

#### 4.4.1 Concept Destructible

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

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

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

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

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

#### 4.4.2 Concept Constructible

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

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

§ 4.4.2
template <class T, class... Args>
concept bool __BindableReference = // exposition only
   is_reference<T>::value && requires(Args&&... args) {
   T(std::forward<Args>(args)...);
};

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

4.4.3 Concept DefaultConstructible [concepts.lib.object.defaultconstructible]

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

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

4.4.4 Concept MoveConstructible [concepts.lib.object.moveconstructible]

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

1 Let rv be an rvalue of type remove_cv_t<T>. Then MoveConstructible<T>() is satisfied if and only if
1.1 After the definition T u = rv;, u is equal to the value of rv before the construction.
1.2 T{rv} or *new T{rv} is equal to the value of rv before the construction.
2 rv's resulting state is unspecified. [Note: rv must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether rv has been moved from or not. —end note]

4.4.5 Concept CopyConstructible [concepts.lib.object.copyconstructible]

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

1 Let v be an lvalue of type (possibly const) remove_cv_t<T> or an rvalue of type const remove_cv_<-
t<T>. Then CopyConstructible<T>() is satisfied if and only if
1.1 After the definition T u = v; v is equal to u.
1.2 T{v} or *new T{v} is equal to v.
4.4.6 Concept Movable

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

4.4.7 Concept Copyable

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

4.4.8 Concept Semiregular

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

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

4.4.9 Concept Regular

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

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

4.5 Callable concepts

4.5.1 In general

The concepts in this section describe the requirements on function objects (5.3) and their arguments.

4.5.2 Concept Invocable

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

2 [Note: Since the `invoke` function call expression is not required to be equality-preserving (4.1.1), a function that generates random numbers may satisfy Invocable. — end note]
4.5.3 Concept RegularInvocable

```cpp
template <class F, class... Args>
concept bool RegularInvocable() {
    return Invocable<F, Args...>();
}
```

The invoke function call expression shall be equality-preserving (4.1.1). [Note: This requirement supersedes the annotation in the definition of Invocable. — end note]

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

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

4.5.4 Concept Predicate

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

4.5.5 Concept Relation

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

```cpp
template <class R, class T, class U>
concept bool Relation() {
    return Relation<R, T>() &&
    Relation<R, U>() &&
    CommonReference<const T&, const U&>() &&
    Relation<R, common_reference_t<const T&, const U&>>() &&
    Predicate<R, T, U>() &&
    Predicate<R, U, T>();
}
```

Let \( r \) be any object of type \( R \), \( a \) be any object of type \( T \), \( b \) be any object of type \( U \), and \( C \) be \( \text{common_reference}_t<const T&, const U&> \). Then \( \text{Relation}<R, T, U>() \) is satisfied if and only if

(1.1) \[ \text{bool}(r(a, b)) = \text{bool}(r(C(a), C(b))). \]

(1.2) \[ \text{bool}(r(b, a)) = \text{bool}(r(C(b), C(a))). \]

4.5.6 Concept StrictWeakOrder

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

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

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

\[(2.1)\quad \text{comp}(a, b) \land \text{comp}(b, c) \implies \text{comp}(a, c)\]

\[(2.2)\quad \text{equiv}(a, b) \land \text{equiv}(b, c) \implies \text{equiv}(a, c)\]

[Note: Under these conditions, it can be shown that]

\[(2.2.1)\quad \text{equiv}\text{ is an equivalence relation}\]

\[(2.2.2)\quad \text{comp}\text{ induces a well-defined relation on the equivalence classes determined by equiv}\]

\[(2.2.3)\quad \text{The induced relation is a strict total ordering. – end note}\]
5 General utilities library

5.1 General

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

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Utility components</td>
<td>&lt;experimental/ranges/utility&gt;</td>
</tr>
<tr>
<td>5.3 Function objects</td>
<td>&lt;experimental/ranges/functional&gt;</td>
</tr>
<tr>
<td>5.4 Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>5.5 Tagged tuple-like types</td>
<td>&lt;experimental/ranges/utility&gt; &amp; &lt;experimental/ranges/tuple&gt;</td>
</tr>
</tbody>
</table>

5.2 Utility components

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

Header <experimental/ranges/utility> synopsis

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.

```cpp
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // 5.2.1, swap:
  namespace {
    constexpr unspecified swap = unspecified ;
  }
  // 5.2.2, exchange:
  template <MoveConstructible T, class U=T>
  requires Assignable<T&, U>()
  T exchange(T& obj, U&& new_val);
  // 5.5.2, struct with named accessors
  template <class T>
  concept bool TagSpecifier() {
    return see below;
  }
  template <class F>
  concept bool TaggedType() {
    return see below;
  }
  template <class Base, TagSpecifier... Tags>
  requires sizeof...(Tags) <= tuple_size<Base>::value
  struct tagged;
```

§ 5.2
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 {
    // 5.5.3, 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...>>;
}

Any entities declared or defined directly in namespace std in header <utility> that are not already defined in namespace std::experimental::ranges::v1 in header <experimental/ranges/utility> are imported with using-declarations (ISO/IEC 14882:2014 §7.3.3). [Example:

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

5.2.1 swap

The name swap denotes a customization point object (3.3.2.3). 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 denoted by E1 and E2, the program is ill-formed with no diagnostic required.

- (1.2) Otherwise, (void)swap_ranges(E1, E2) if E1 and E2 are lvalues of array types (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 denoted values. ranges::swap(E1, E2) is a constant expression if the constructor selected by overload resolution for T{std::move(E1)} is a constexpr constructor and the expression E1 = std::move(E2) can appear in a constexpr function. noexcept(ranges::swap(E1, E2)) is equal to is_nothrow_move_constructible<T>::value && is_nothrow_move_assignable<T>::value. If either MoveConstructible or Assignable is not satisfied, the program is ill-formed with no diagnostic required.

§ 5.2.1
(1.4) Otherwise, `ranges::swap(E1, E2)` is ill-formed.

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

5.2.2 exchange

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

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

5.3 Function objects

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

```
namespaces std { namespace experimental { namespace ranges { inline namespace v1 {

  // 5.3.1, invoke:
  template <class F, class... Args>
  result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

  // 5.3.2, comparisons:
  template <class T = void>
  requires EqualityComparable<T>() || Same<T, void>()
  struct equal_to;

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

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

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

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

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

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

§ 5.3
// 5.3.3, identity:
struct identity;
}}}

Any entities declared or defined directly in namespace std in header <functional> that are not already defined in namespace std::experimental::ranges in header <experimental/ranges/functional> are imported with using-declarations (ISO/IEC 14882:2014 §7.3.3). [Example:

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    using std::reference_wrapper;
    using std::ref;
    // ... others
}}}}
— end example]

Any nested namespaces defined directly in namespace std in header <functional> that are not already defined in namespace std::experimental::ranges in header <experimental/ranges/functional> are aliased with a namespace-alias-definition (ISO/IEC 14882:2014 §7.3.2). [Example:

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    namespace placeholders = std::placeholders;
}}}}
— end example]

5.3.1 Function template invoke

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

Effects: Equivalent to:
return INVOKE(std::forward<F>(f), std::forward<Args>(args)...); (ISO/IEC14882:2014 §20.9.2).

5.3.2 Comparisons

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).

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

operator() returns x == y.

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

operator() returns x != y.

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

§ 5.3.2
constexpr bool operator()(const T& x, const T& y) const;
operator() returns \textit{x} > \textit{y}.

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

template <class T = void>
requires StrictTotallyOrdered<T>() || Same<T, void>()
struct greater_equal {
    constexpr bool operator()(const T& x, const T& y) const;
}
operator() returns \textit{x} \geq \textit{y}.

template <class T = void>
requires StrictTotallyOrdered<T>() || Same<T, void>()
struct less_equal {
    constexpr bool operator()(const T& x, const T& y) const;
}
operator() returns \textit{x} \leq \textit{y}.

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

typedef unspecified is_transparent;
}
operator() returns std::forward<T>(\textit{t}) == std::forward<U>(\textit{u}).

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

typedef unspecified is_transparent;
}
operator() returns std::forward<T>(\textit{t}) != std::forward<U>(\textit{u}).

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

typedef unspecified is_transparent;
};
operator() returns `std::forward<T>(t) > std::forward<U>(u)`.

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

    typedef unspecified is_transparent;
};
```

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

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

    typedef unspecified is_transparent;
};
```

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

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

    typedef unspecified is_transparent;
};
```

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

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

### 5.3.3 Class identity

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

    typedef unspecified is_transparent;
};
```

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

### 5.4 Metaprogramming and type traits

#### 5.4.1 Header `<type_traits>` synopsis

[Editor's note: Change the `<type_traits>` synopsis (ISO/IEC 14882:2014 §20.10.2) as follows. Note: this change is intended to be made in namespace std.]

```cpp
namespace std {
    [...]
    // 20.10.4.3, type properties:
```
[...] template <class T> struct is_move_assignable;

template <class T, class U> struct is_swappable_with;
template <class T> struct is_swappable;

template <class T> struct is_destructible;
[...]
template <class T> struct is_nothrow_move_assignable;

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

template <class T> struct is_nothrow_destructible;
[...]

// 20.10.7.6, 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> struct underlying_type;
[...]
template <class... T>
  using common_type_t = typename common_type<T...>::type;
template <class... T>
  using common_reference_t = typename common_reference<T...>::type;
template <class T>
  using underlying_type_t = typename underlying_type<T>::type;
[...]

// 20.15.4.3, type properties
[...]
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;
[...]
}

5.4.2 Type properties [meta.unary.prop]
[Editor’s note: Change [meta.unary.prop], Table 49 — “Type property predicates” in ISO/IEC 14882:2014 as indicated. The following is taken from the current Working Draft of C++17.]

Table 49 — Type property predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
§ 5.4.2                                        |                                               | 30            |
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T, class U&gt;</td>
<td>The expressions swap(declval&lt;T&gt;(), declval&lt;U&gt;()) and swap(declval&lt;T&gt;(), declval&lt;U&gt;()) are each well-formed when treated as an unevaluated operand in an overload-resolution context for swappable values (4.2.11). 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. Remark: 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.</td>
<td>T and U shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>struct is_swappable_with;</td>
<td>For a referenceable type T, the same result as is_swappable_with_v&lt;T&amp;,T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, (possibly cv-qualified) void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template &lt;class T&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>struct is_swappable;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class T, class U&gt;</td>
<td>is_swappable_with_v&lt;T, U&gt; is true and each swap expression of the definition of is_swappable_with&lt;T, U&gt; is known not to throw any exceptions (ISO/IEC 14882:2014 §5.3.7).</td>
<td>T and U shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td>For a referenceable type T, the same result as</td>
<td>T shall be a complete type, (possibly cv-qualified)</td>
</tr>
<tr>
<td>struct is_nothrow_swappable;</td>
<td>is_nothrow_swappable_ with_v&lt;T&amp;, T&amp;&gt;, otherwise false.</td>
<td>void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>

5.4.3 Other transformations

[meta.trans.other]

[Editor’s note: Change Table 57 – “Other Transformations” in ISO/IEC 14882:2014 as follows:]

Table 57 — Other transformations

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class... T&gt;</td>
<td></td>
<td>The member typedef type shall be defined or omitted as specified below.</td>
</tr>
<tr>
<td>struct common_type;</td>
<td></td>
<td>If it is omitted, there shall be no member type. Each type in the parameter pack T shall be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complete or (possibly cv) void. A program may specialize this trait if at least one template</td>
</tr>
<tr>
<td></td>
<td></td>
<td>parameter in the specialization depends on a user-defined type and sizeof...(T) == 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remark: Such specializations are needed when only explicit conversions are desired among the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>template arguments.</td>
</tr>
<tr>
<td>template &lt;class T, class U, class TQual, class UQual&gt;</td>
<td></td>
<td>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. Remark: Such specializations may be used to influence the result of common_reference.</td>
</tr>
<tr>
<td>struct basic_common_reference;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt;class... T&gt;</td>
<td></td>
<td>The member typedef type shall be defined or omitted as specified below.</td>
</tr>
<tr>
<td>struct common_reference;</td>
<td></td>
<td>If it is omitted, there shall be no member type. Each type in the parameter pack T shall be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complete or (possibly cv) void.</td>
</tr>
</tbody>
</table>

§ 5.4.3
[Editor’s note: Delete [meta.trans.other]/p3 from ISO/IEC 14882:2014 and replace it with the following:]

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

\[
\begin{align*}
(1.1) & \quad \text{If } A \text{ and } B \text{ are both lvalue reference types, } \text{COMMON_REF}(A, B) \text{ is } \text{COND_RES}(\text{COPYCV}(X, Y) &, \text{COPYCV}(Y, X) &). \\
(1.2) & \quad \text{Otherwise, let } C \text{ be } \text{RREF_RES}(\text{COMMON_REF}(X&, Y&)). \text{ If } A \text{ and } B \text{ are both rvalue reference types, and } C \text{ is well-formed, and } \text{is_convertible}<A, C>::value \text{ and } \text{is_convertible}<B, C>::value \text{ are true, then } \text{COMMON_REF}(A, B) \text{ is } C. \\
(1.3) & \quad \text{Otherwise, let } D \text{ be } \text{COMMON_REF}(\text{const X&, Y&}). \text{ If } A \text{ is an rvalue reference and } B \text{ is an lvalue reference and } D \text{ is well-formed and } \text{is_convertible}<A, D>::value \text{ is true, then } \text{COMMON_REF}(A, B) \text{ is } D. \\
(1.4) & \quad \text{Otherwise, if } A \text{ is an lvalue reference and } B \text{ is an rvalue reference, then } \text{COMMON_REF}(A, B) \text{ is } \text{COMMON-_REF}(B, A). \\
(1.5) & \quad \text{Otherwise, } \text{COMMON_REF}(A, B) \text{ is } \text{decay_t<COND_RES(\text{CREF}(A), \text{CREF}(B))>}. \\
\end{align*}
\]

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

2 For the \textit{common_type} trait applied to a parameter pack \( T \) of types, the member \textit{type} shall be either defined or not present as follows:

\[
\begin{align*}
(2.1) & \quad \text{If } \text{sizeof}...(T) \text{ is zero, there shall be no member type.} \\
(2.2) & \quad \text{Otherwise, if } \text{sizeof}...(T) \text{ is one, let } T_0T_1 \text{ denote the sole type in the pack } T. \text{ The member typedef } \textit{type} \text{ shall denote the same type as } \text{decay_t<T_0T_1>.} \\
(2.3) & \quad \text{Otherwise, if } \text{sizeof}...(T) \text{ is two, let } T_1 \text{ and } T_2 \text{ denote the two types in the pack } T, \text{ and let } D_1 \text{ and } D_2 \text{ be } \text{decay_t<T_1>} \text{ and } \text{decay_t<T_2>} \text{ respectively. Then} \\
(2.3.1) & \quad \text{If } D_1 \text{ and } T_1 \text{ denote the same type and } D_2 \text{ and } T_2 \text{ denote the same type, then} \\
(2.3.1.1) & \quad \text{If } \text{COMMON_REF}(T_1, T_2) \text{ is well-formed, then the member typedef } \textit{type} \text{ denotes that type.} \\
(2.3.1.2) & \quad \text{Otherwise, there shall be no member type.} \\
(2.3.2) & \quad \text{Otherwise, if } \text{common_type_t<D_1, D_2>} \text{ is well-formed, then the member typedef } \textit{type} \text{ denotes that type.} \\
(2.3.3) & \quad \text{Otherwise, there shall be no member type.} \\
(2.4) & \quad \text{Otherwise, if } \text{sizeof}...(T) \text{ is greater than one, let } T_1, T_2, \text{ and } \text{rest} \text{, respectively, denote the first, second, and (pack of) remaining types comprising } T. \text{ [Note: } \text{sizeof}...(\text{rest}) \text{ may be zero. —end note] Let } C \text{ denote the type, if any, of an unevaluated conditional expression (5.16) whose first operand is an arbitrary value of type bool whose second operand is an xvalue of type } T_1, \text{ and whose third operand is an xvalue of type } T_2\text{ be the type } \text{common_type_t<T_1, T_2}. \text{ Then} \\
(2.4.1) & \quad \text{If there is such a type } C, \text{ the member typedef } \textit{type} \text{ shall denote the same type, if any, as } \text{common_type_t<C, rest>}. \\
\]
— Otherwise, there shall be no member type.

3 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:

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

(3.2) — Otherwise, if sizeof...(T) is one, let $T_1$ denote the sole type in the pack $T$. The member typedef type shall denote the same type as $T_1$.

(3.3) — Otherwise, if sizeof...(T) is two, let $T_1$ and $T_2$ denote the two types in the pack $T$. Then

(3.3.1) — If COMMON_REF($T_1$, $T_2$) is well-formed and denotes a reference type then the member typedef type denotes that type.

(3.3.2) — Otherwise, if basic_common_reference<UNCVREF($T_1$), UNCVREF($T_2$), XREF($T_1$), XREF($T_2$)>::type is well-formed, then the member typedef type denotes that type.

(3.3.3) — Otherwise, if common_type_t<$T_1$, $T_2$> is well-formed, then the member typedef type denotes that type.

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

(3.4) — Otherwise, if sizeof...(T) is greater than two, let $T_1$, $T_2$, and Rest, respectively, denote the first, second, and (pack of) remaining types comprising $T$. Let $C$ be the type common_reference_t<$T_1$, $T_2$>. Then:

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

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

5.5 Tagged tuple-like types [taggedtup]

5.5.1 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).

5.5.2 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 (5.5.4) and tagged_tuple (5.5.5).

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

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

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

    template <class T>
    concept bool TagSpecifier() {
        return implementation-defined;
    }

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

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

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

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

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

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

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

§ 5.5.2
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 `$TAGELEM(F)$` name the argument type of the `$TaggedType F$`.

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

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

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

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

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

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

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

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

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

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

14. `Returns:` `*this`.

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

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

16. `Returns:` `*this`.

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

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

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

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

19. `Returns:` `*this`.

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

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

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

§ 5.5.2
Effects: Calls swap on the result of applying static_cast to *this and that.

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

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

Remarks: The expression in the noexcept is equivalent to:

noexcept(lhs.swap(rhs))

Effects: Equivalent to lhs.swap(rhs).

5.5.3 Tuple-like access to tagged

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> { ];
}

5.5.4 Alias template tagged_pair

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

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

[Example:

// See 7.2:
tagged_pair<tag::min(int), tag::max(int)> p{0, 1};
assert(&p.min() == &p.first);
assert(&p.max() == &p.second);

— end example]

5.5.4.1 Tagged pair creation functions

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 [ Example:

// 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<T1, Tag1, Tag2>.

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

3 [Example: In place of:

\[
\text{return tagged_pair<tag::min(int), tag::max(double)>(5, 3.1415926)}; \quad // \text{explicit types}
\]
a C++ program may contain:

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

—end example]

5.5.5 Alias template tagged_tuple

[tagged.tuple]

Header `<experimental/ranges/tuple>` synopsis

```cpp
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);

}}}}
```

Any entities declared or defined in namespace `std` in header `<tuple>` that are not already defined in namespace `std::experimental::ranges` in header `<experimental/ranges/tuple>` are imported with `using-declarations` (ISO/IEC 14882:2014 §7.3.3). [Example:

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

—end example]

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

[Example:

// See 7.2:

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]

5.5.5.1 Tagged tuple creation functions

[tagged.tuple.creation]

```cpp
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)...).` 

[Example:

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

§ 5.5.5.1
creates a tagged tuple of type

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

— end example ]
6 Iterators library

6.1 General

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), stream buffers (ISO/IEC 14882:2014 §27.6), and ranges (6.9).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 5.

Table 5 — Iterators library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>6.6</td>
<td>Iterator primitives</td>
</tr>
<tr>
<td>6.7</td>
<td>Predefined iterators</td>
</tr>
<tr>
<td>6.8</td>
<td>Stream iterators</td>
</tr>
<tr>
<td>6.9</td>
<td>Ranges</td>
</tr>
</tbody>
</table>

6.2 Iterator requirements

6.2.1 In general

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.

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 International Standard defines five categories of iterators, according to the operations defined on them: input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators, as shown in Table 6.

Table 6 — Relations among iterator categories

| Random Access | Bidirectional | Forward | Input | Output |

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.

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

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

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called **past-the-end** values. Values of an iterator \( i \) for which the expression \( *i \) is defined are called dereferenceable. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any sequence. [Example: After the declaration of an uninitialized pointer \( x \) (as with \texttt{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.

Most of the library’s algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a **sentinel** that designate the beginning and end of the computation. An iterator and a sentinel denoting a range are comparable. A sentinel denotes an element when it compares equal to an iterator \( i \), and \( i \) points to that element. The types of a sentinel and an iterator that denote a range must satisfy **Sentinel (6.2.7)**.

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 \( i \), they denote a range.

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

A range \([i,s)\) is valid if and only if \( s \) is reachable from \( i \). The result of the application of functions in the library to invalid ranges is undefined.

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

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

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

### 6.2.2 Concept Readable

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

```cpp
#include <type_traits>

template <class I>
class concept bool Readable() {
  return Movable<I>() && DefaultConstructible<I>() &&
  requires(const I& i) {  // returns a value_type_t<
    typename value_type_t<I>;
    typename reference_t<I>;
    typename rvalue_reference_t<I>;
    { *i } -> Same<reference_t<I>>;
  }
}
```

3) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.
A Readable type has an associated value type that can be accessed with the `value_type_t` alias template.

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

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

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

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

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

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

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

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

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

When instantiated with a type `I` 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]

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

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

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

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

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

After evaluating the assignment expression, \( o \) is not required to be dereferenceable.

If \( E \) is an xvalue (ISO/IEC 14882:2014 §3.10), the resulting state of the object it denotes is unspecified. [Note: The object must still meet the requirements of any library component that is using it. The operations listed in those requirements must work as specified whether the object has been moved from or not. — end note]

[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]

6.2.4 Concept WeaklyIncrementable

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

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

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

1. **++i** is valid if and only if **i++** is valid.

2. If \( i \) is incrementable, then both **++i** and **i++** advance \( i \) to the next element.

3. If \( i \) is incrementable, then `&++i == &i`.

[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]
6.2.5 Concept Incrementable  

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

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

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

\[(2.1) \quad \text{If } bool(a == b) \text{ then } bool(a++ == b)\].
\[(2.2) \quad \text{If } bool(a == b) \text{ then } bool((a++, a) == ++b)\].

[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]

6.2.6 Concept Iterator  

The **Iterator** concept forms the basis of the iterator concept taxonomy; every iterator satisfies the **Iterator** requirements. This concept specifies operations for dereferencing and incrementing an iterator. Most algorithms will require additional operations to compare iterators with sentinels (6.2.7), to read (6.2.9) or write (6.2.10) values, or to provide a richer set of iterator movements (6.2.11, 6.2.12, 6.2.13).

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

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

6.2.7 Concept Sentinel  

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

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

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

§ 6.2.7
The domain of $\mathbf{==}$ can change over time. Given an iterator $i$ and sentinel $s$ such that $[i, s)$ denotes a range and $i \neq s$, $[i, s)$ 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.

6.2.8 Concept SizedSentinel

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

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

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

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 $\text{bool}(i == s)$ be true. SizedSentinel$<S, I>()$ is satisfied if and only if:

1. If $N$ is representable by $\text{difference_type_t}<I>$, then $s - i$ is well-defined and equals $N$.
2. If $-N$ is representable by $\text{difference_type_t}<I>$, then $i - s$ is well-defined and equals $-N$.

[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 (3.3.1.3). — end note]

[Note: The SizedSentinel concept is satisfied by pairs of RandomAccessIterators (6.2.13) and by counted iterators and their sentinels (6.7.6.1). — end note]

6.2.9 Concept InputIterator

The InputIterator concept is a refinement of Iterator (6.2.6). It defines requirements for a type whose referenced values can be read (from the requirement for Readable (6.2.2)) and which can be both pre- and post-incremented. [Note: Unlike in ISO/IEC 14882, input iterators are not required to satisfy EqualityComparable (4.3.3). — end note]

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

§ 6.2.9
6.2.10 Concept OutputIterator

The OutputIterator concept is a refinement of Iterator (6.2.6). It defines requirements for a type that can be used to write values (from the requirement for Writable (6.2.3)) and which can be both pre- and post-incremented. However, output iterators are not required to satisfy EqualityComparable.

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

[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]

6.2.11 Concept ForwardIterator

The ForwardIterator concept refines InputIterator (6.2.9), adding equality comparison and the multi-pass guarantee, specified below.

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

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

\( a == b \) implies \( ++a == ++b \)

\( (\{ (X x)\{++x;\}(a), *a\}) \) is equivalent to the expression \( *a \).

4 [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]

6.2.12 Concept BidirectionalIterator

The BidirectionalIterator concept refines ForwardIterator (6.2.11), and adds the ability to move an iterator backward as well as forward.

```
template <class I>
concept bool BidirectionalIterator() {
    return ForwardIterator<I>() &&
            DerivedFrom<iterator_category_t<I>, bidirectional_iterator_tag>() &&
            requires(I i) {
```
A bidirectional iterator \( r \) is decrementable if and only if there exists some \( s \) such that \( ++s == r \). The expressions \( --r \) and \( r-- \) are only valid if \( r \) is decrementable.

Let \( a \) and \( b \) be decrementable objects of type \( I \). Then \texttt{BidirectionalIterator\langle I\rangle()} is satisfied if and only if:

1. \( k--a == &a \).
2. If \( \text{bool}(a == b) \), then \( \text{bool}(a-- == b) \).
3. If \( \text{bool}(a == b) \), then \( \text{bool}((a--, a) == --b) \).
4. If \( a \) is incrementable and \( \text{bool}(a == b) \), then \( \text{bool}(-((++a) == b)) \).
5. If \( \text{bool}(a == b) \), then \( \text{bool}(++(--a) == b) \).

### 6.2.13 Concept RandomAccessIterator

The \texttt{RandomAccessIterator} concept refines \texttt{BidirectionalIterator} (6.2.12) and adds support for constant-time advancement with \( +=, +, -=, \) and \( - \), and the computation of distance in constant time with \( - \). Random access iterators also support array notation via subscripting.

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

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

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

6.3 Indirect callable requirements

6.3.1 In general

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

6.3.2 Indirect callables

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

```cpp
template <class F>
caption bool IndirectInvocable() {
    return Invocable<F>();
}

template <class F, class I>
caption bool IndirectInvocable() {
    return Readable<I>() &&
        Invocable<F, value_type_t<I>>(() &&
        Invocable<F, reference_t<I>>(() &&
        Invocable<F, iter_common_reference_t<I>>(();
    }

template <class F, class I1, class I2>
caption bool IndirectInvocable() {
    return Readable<I1>() && Readable<I2>() &&
        Invocable<F, value_type_t<I1>, value_type_t<I2>>(() &&
        Invocable<F, value_type_t<I1>, reference_t<I2>>(() &&
        Invocable<F, reference_t<I1>, value_type_t<I2>>(() &&
        Invocable<F, reference_t<I1>, reference_t<I2>>(() &&
        Invocable<F, iter_common_reference_t<I1>, iter_common_reference_t<I2>>(();
    }

template <class F>
caption bool IndirectRegularInvocable() {
    return RegularInvocable<F>();
}

template <class F, class I>
caption bool IndirectRegularInvocable() {
    return Readable<I>() &&
        RegularInvocable<F, value_type_t<I>>(() &&
        RegularInvocable<F, reference_t<I>>(() &&
        RegularInvocable<F, iter_common_reference_t<I>>(();
    }

template <class F, class I1, class I2>
caption bool IndirectRegularInvocable() {
    return Readable<I1>() && Readable<I2>() &&
        RegularInvocable<F, value_type_t<I1>, value_type_t<I2>>(() &&
```
template <class F, class I>
concept bool IndirectPredicate() {
    return Readable<I>() &&
    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>
concept bool IndirectPredicate() {
    return Readable<I1>() && Readable<I2>() &&
    Predicate<F, value_type_t<I1>, value_type_t<I2>>() &&
    Predicate<F, value_type_t<I1>, reference_t<I2>>() &&
    Predicate<F, value_type_t<I1>, reference_t<I2>>() &&
    Predicate<F, reference_t<I1>, value_type_t<I2>>() &&
    Predicate<F, reference_t<I1>, reference_t<I2>>() &&
    Predicate<F, iter_common_reference_t<I1>, iter_common_reference_t<I2>>();
}

template <class F, class I1, class I2 = I1>
concept bool IndirectRelation() {
    return Readable<I1>() && Readable<I2>() &&
    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() {
    return Readable<I1>() && Readable<I2>() &&
    StrictWeakOrder<F, value_type_t<I1>, value_type_t<I2>>() &&
    StrictWeakOrder<F, value_type_t<I1>, reference_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 { };
6.3.3 Class template projected

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

\begin{verbatim}
template <Readable I, IndirectRegularInvocable<I> Proj>
struct projected {
    using value_type = decay_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>;
};
\end{verbatim}

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

6.4 Common algorithm requirements

6.4.1 In general

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

[Note: The equal_to<> and less<> (5.3.2) 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 EqualityComparable (4.3.3), and less<> requires its arguments satisfy StrictTotallyOrdered (4.3.4). — end note]

6.4.2 Concept \textit{IndirectlyMovable}

The \textit{IndirectlyMovable} concept specifies the relationship between a Readable type and a Writable type between which values may be moved.

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

The \textit{IndirectlyMovableStorable} concept augments \textit{IndirectlyMovable} with additional requirements enabling the transfer to be performed through an intermediate object of the Readable type’s value type.

\begin{verbatim}
template <class In, class Out>
concept bool IndirectlyMovableStorable() {
    return IndirectlyMovable<In, Out>() &&
    Writable<Out, value_type_t<In>>() &&
    Movable<value_type_t<In>>() &&
    Constructible<value_type_t<In>>, rvalue_reference_t<In>>() &&
    Assignable<value_type_t<In>&, rvalue_reference_t<In>>() &&
};
\end{verbatim}
6.4.3 Concept IndirectlyCopyable

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

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

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

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

6.4.4 Concept IndirectlySwappable

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

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

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

6.4.5 Concept IndirectlyComparable

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

```cpp
template <class I1, class I2, class R = equal_to<> , class P1 = identity, class P2 = identity>
concept bool IndirectlyComparable() {
    return IndirectRelation<R, projected<I1, P1>, projected<I2, P2>>()
}
```
6.4.6 Concept Permutable [commonalgoreq.permutable]
1 The Permutable concept specifies the common requirements of algorithms that reorder elements in place by moving or swapping them.

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

6.4.7 Concept Mergeable [commonalgoreq.mergeable]
1 The Mergeable concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by copying elements.

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

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

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

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

```cpp
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    // 6.6, primitives:
    // exposition only
    template <class T> concept bool _Dereferenceable = requires(T& t) { {*t} -> auto&&; };

    // 6.6.1, utilities:
    namespace {
        // 6.6.1.1, iter_move:
        constexpr unspecified iter_move = unspecified;

        // 6.6.1.2, iter_swap:
        constexpr unspecified iter_swap = unspecified;
    }

    // 6.6.2, traits:

}}}
```
template <class> struct difference_type;
template <class> struct value_type;
template <class> struct iterator_category;

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

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

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

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 &>()));

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

template <class I1, class I2> struct is_indirectly_movable;
template <class I1, class I2 = I1> struct is_indirectly_swappable;
template <class I1, class I2> struct is_nothrow_indirectly_movable;
template <class I1, class I2 = I1> struct is_nothrow_indirectly_swappable;

template <class I1, class I2> constexpr bool is_indirectly_movable_v
    = is_indirectly_movable<I1, I2>::value;

template <class I1, class I2> constexpr bool is_indirectly_swappable_v
    = is_indirectly_swappable<I1, I2>::value;

template <class I1, class I2> constexpr bool is_nothrow_indirectly_movable_v
    = is_nothrow_indirectly_movable<I1, I2>::value;

template <class I1, class I2> constexpr bool is_nothrow_indirectly_swappable_v
    = is_nothrow_indirectly_swappable<I1, I2>::value;

template <class Iterator> using iterator_traits = see below;

// 6.6.4, 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 { };

// 6.6.5, iterator operations:
template <Iterator I>
    void advance(I &, difference_type_t<I> n);

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

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

template <Iterator I, Sentinel<I> S>
    difference_type_t<I> distance(I first, S last);
template <Iterator I>
    I next(I x, difference_type_t<I> n = 1);
template <Iterator I, Sentinel<I> S>
    I next(I x, S bound);
template <Iterator I, Sentinel<I> S>
    I next(I x, difference_type_t<I> n, S bound);
template <BidirectionalIterator I>
    I prev(I x, difference_type_t<I> n = 1);
template <BidirectionalIterator I>
    I prev(I x, difference_type_t<I> n, I bound);

// 6.7, predefined iterators and sentinels:

// 6.7.1, reverse iterators:
template <BidirectionalIterator I> class reverse_iterator;

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

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

// 6.7.2, insert iterators:
template <BidirectionalIterator I>
reverse_iterator<I> make_reverse_iterator(I i);

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);

// 6.7.3, move iterators and sentinels:
template <InputIterator I> class move_iterator;

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

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

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

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

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

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

template <class I1, class I2>
requires SizedSentinel<I1, I2>()
difference_type_t<I2> operator-(
const move_iterator<I1>& x,
const move_iterator<I2>& y);

template <RandomAccessIterator I>
move_iterator<I>
operator+(difference_type_t<I> n,
const move_iterator<I>& x);
template <InputIterator I>
    move_iterator<I> make_move_iterator(I i);

template <Semiregular S> class move_sentinel;

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

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

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

    // 6.7.4, 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);
        bool operator==(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
        bool operator==(const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
        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);

// 6.7.5, default sentinels:
class default_sentinel;

// 6.7.6, counted iterators:
template <Iterator I> class counted_iterator;

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

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

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

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

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

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

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

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

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

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

template <Iterator I>
§ 6.5
counted_iterator<I> make_counted_iterator(I i, difference_type_t<I> n);

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

// 6.7.8, 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;

// 6.7.7, dangling wrapper:
template <class T> class dangling;

// 6.8, 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;

// 6.10, 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;
}

// 6.11, range primitives:
namespace {
    constexpr unspecified size = unspecified;
    constexpr unspecified empty = unspecified;
    constexpr unspecified data = unspecified;
    constexpr unspecified cdata = unspecified;
}

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

template <SizedRange R>
difference_type_t<iterator_t<R>> distance(R&& r);
}}}

namespace std {
    // 6.6.3, iterator traits:
    template <experimental::ranges::Iterator I>
    struct iterator_traits;
    template <experimental::ranges::InputIterator I>
    struct iterator_traits;
    template <experimental::ranges::InputIterator I>
    requires Sentinel<I, I>()
    struct iterator_traits;
}

1 Any entities declared or defined in namespace std in header <iterator> that are not already defined in namespace std::experimental::ranges in header <experimental/ranges/iterator> are imported with
using-declarations (ISO/IEC 14882:2014 §7.3.3).

6.6 Iterator primitives

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

6.6.1 Iterator utilities

6.6.1.1 iter_move

The name iter_move denotes a customization point object (3.3.2.3). The effect of the expression ranges::iter_move(E) for some expression E is equivalent to iter_move(E), with overload resolution (ISO/IEC 14882:2014 §13.3) performed in a context described below.

The context in which the expression iter_move(E) is evaluated

— does not include a declaration of ranges::iter_move,

— includes the lookup set produced by argument-dependent lookup (ISO/IEC 14882:2014 §3.4.2), and

— includes an iter_move function template equivalent to the following:

```cpp
// exposition only
template <_Dereferenceable I>
    see below iter_move(I&& r) noexcept(see below);
```

The return type is Ret where Ret is remove_reference_t<reference_t<I>>&& if reference_t<I> is a reference type; decay_t<reference_t<I>>, otherwise.

The expression in the noexcept is equivalent to:

```
    noexcept(static_cast<Ret>(*r))
```

Effects: Equivalent to: return static_cast<Ret>(*r);

If, for the iter_move function selected by overload resolution, iter_move(E) does not equal *E, the program is ill-formed with no diagnostic required.

6.6.1.2 iter_swap

The name iter_swap denotes a customization point object (3.3.2.3). The effect of the expression ranges::iter_swap(E1, E2) for some expressions E1 and E2 is equivalent to (void)iter_swap(E1, E2), with overload resolution (ISO/IEC 14882:2014 §13.3) performed in a context described below.

The context in which the expression (void)iter_swap(E1, E2) is evaluated

— does not include a declaration of ranges::iter_swap,

— includes the lookup set produced by argument-dependent lookup (ISO/IEC 14882:2014 §3.4.2), and

— includes iter_swap function template(s) equivalent to the following:

```cpp
// exposition only
template <class I1, class I2>
    void iter_swap(I1&&, I2&&) = delete;
```

```
// exposition only
template <class I1, class I2,
    Readable _R1 = remove_reference_t<I1>,
    Readable _R2 = remove_reference_t<I2>>
    requires Swappable<reference_t<_R1>, reference_t<_R2>>()
    void iter_swap(I1&& r1, I2&& r2)
        noexcept(is_nothrow_swappable_with_v<reference_t<_R1>, reference_t<_R2>>);
```
Effects: Equivalent to ranges::swap(*r1, *r2).

```cpp
template <class I1, class I2,
    Readable _R1 = std::remove_reference_t<I1>,
    Readable _R2 = std::remove_reference_t<I2>>
    requires !Swappable<reference_t<_R1>, reference_t<_R2>>()
    && IndirectlyMovableStorable<_R1, _R2>()
    && IndirectlyMovableStorable<_R2, _R1>()
void iter_swap(I1&& r1, I2&& r2)
    noexcept(is_nothrow_indirectly_movable_v<_R1, _R2>
    && is_nothrow_indirectly_movable_v<_R2, _R1>);
```

Effects: Exchanges values referred to by two Readable objects.

[Example: Below is a possible implementation:

```cpp```
  value_type_t<_R1> tmp(iter_move(r1));
  *r1 = iter_move(r2);
  *r2 = std::move(tmp);
```cpp```

—end example]

If the iter_swap function selected by overload resolution does not swap the values denoted by the expressions E1 and E2, the program is ill-formed with no diagnostic required.

6.6.2 Iterator traits

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 (6.2.4), R is the name of a type that satisfies the Readable concept (6.2.2), and II is the name of a type that satisfies the InputIterator concept (6.2.9) 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.

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

```cpp```
```
§ 6.6.2 61
```
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;

3 Users may specialize difference_type on user-defined types.

4 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;

5 Users may specialize iterator_category on user-defined types.

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

  (6.1) — 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.

  (6.2) — 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.

  (6.3) — 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.

  (6.4) — 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.

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

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

7 rvalue_reference_t<T> is implemented as if:

§ 6.6.2
template <_Dereferenceable T>
    requires see below using rvalue_reference_t
    = decltype(ranges::iter_move(declval<T&>()));

The expression in the requires clause is equivalent to:

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

The class templates is_indirectly_movable, is_nothrow_indirectly_movable, is_indirectly_swappable, and is_nothrow_indirectly_swappable shall be defined as follows:

```
template <class In, class Out>
struct is_indirectly_movable : false_type { };  // exposition only

template <class In, class Out>
requires IndirectlyMovable<In, Out>()
struct is_indirectly_movable<In, Out> : true_type { };  

```

```
template <class In, class Out>
requires IndirectlyMovable<In, Out>()
struct is_nothrow_indirectly_movable<In, Out> : false_type { };  // exposition only

```

```
template <class In, class Out>
requires IndirectlyMovable<In, Out>()
struct is_nothrow_indirectly_movable<In, Out> :
    std::integral_constant<bool,
        is_nothrow_constructible<value_type_t<In>, rvalue_reference_t<In>>::value &&
        is_nothrow_assignable<value_type_t<In> &, rvalue_reference_t<In>>::value &&
        is_nothrow_assignable<reference_t<Out>, rvalue_reference_t<In>>::value &&
        is_nothrow_assignable<reference_t<Out>, value_type_t<In>>::value
    } { };  // exposition only

```

```
template <class I1, class I2 = I1>
struct is_indirectly_swappable : false_type { };  // exposition only

```

```
template <class I1, class I2>
requires IndirectlySwappable<I1, I2>()
struct is_indirectly_swappable<I1, I2> : true_type { };  

```

```
template <class I1, class I2 = I1>
requires IndirectlySwappable<I1, I2>()
struct is_nothrow_indirectly_swappable<I1, I2> : false_type { };  // exposition only

```

```
template <class I1, class I2>
requires IndirectlySwappable<I1, I2>()
struct is_nothrow_indirectly_swappable<I1, I2> :
    std::integral_constant<bool,
        noexcept(ranges::iter_swap(declval<I1&>(), declval<I2&>())) &&
        noexcept(ranges::iter_swap(declval<I2&>(), declval<I1&>())) &&
        noexcept(ranges::iter_swap(declval<I1&>(), declval<I1&>())) &&
        noexcept(ranges::iter_swap(declval<I2&>(), declval<I2&>()))
    } { };  // exposition only

```

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 = iterator_category_t<I>;
};
template <InputIterator I> struct __iterator_traits<I> {    // exposition only
    using difference_type = difference_type_t<I>;
    using value_type = value_type_t<I>;
    using reference = reference_t<I>;
    using pointer = typename __pointer_type<I>::type;
    using iterator_category = iterator_category_t<I>;
};
template <class I>
    using iterator_traits = __iterator_traits<I>;

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

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

    template <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]

6.6.3 Standard iterator traits [iterator.stdtraits]

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;
        };

§ 6.6.3
The nested type `value_type` is computed as follows:

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

The nested type `reference` is computed as follows:

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

The nested type `pointer` is computed as follows:

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

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

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

The nested type `reference` is computed as follows:

- (5.1) If `In::reference` is valid and denotes a type, then `std::iterator_traits<In>::reference` is `In::reference`.
- (5.2) Otherwise, `std::iterator_traits<In>::reference` is `experimental::ranges::reference_t<In>`.

The nested type `pointer` is computed as follows:

- (6.1) If `In::pointer` is valid and denotes a type, then `std::iterator_traits<In>::pointer` is `In::pointer`.
- (6.2) Otherwise, `std::iterator_traits<In>::pointer` is `experimental::ranges::iterator_traits<In>::pointer`.

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`.  

§ 6.6.3
Otherwise, if `experimental::ranges::reference_t<In>` is not a reference type, `std::iterator_traits<In>::iterator_category` is `std::input_iterator_tag`.

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

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

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

Otherwise, `std::iterator_traits<In>::iterator_category` is `std::input_iterator_tag`.

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

6.6.4 Standard iterator tags

It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces `category tag` classes which can be used as compile time tags for algorithm selection. [Note: The preferred way to dispatch to more specialized algorithm implementations is with concept-based overloading. —end note] The category tags are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every input iterator of type `I`, `iterator_category_t<I>` shall be defined to be the most specific category tag that describes the iterator’s behavior.

```cpp
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  struct output_iterator_tag { };  
  struct input_iterator_tag { };  
  struct forward_iterator_tag : input_iterator_tag { };  
  struct bidirectional_iterator_tag : forward_iterator_tag { };  
  struct random_access_iterator_tag : bidirectional_iterator_tag { }; 
}}}}
```

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

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

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

—end example]
6.6.5 Iterator operations

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

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

requires: n shall be negative only for bidirectional iterators.

effects: For random access iterators, equivalent to i += n. Otherwise, increments (or decrements for negative n) iterator i by n.

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

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

effects:

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

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

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

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

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

effects:

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

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

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

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

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

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

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

effects: If SizedSentinel<S, I>() is satisfied, returns (last − first); otherwise, returns the number of increments needed to get from first to last.

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

effects: Equivalent to: advance(x, n); return x;
template <Iterator I, Sentinel<I> S>
  I next(I x, S bound);
  Effects: Equivalent to: advance(x, bound); return x;

template <Iterator I, Sentinel<I> S>
  I next(I x, difference_type_t<I> n, S bound);
  Effects: Equivalent to: advance(x, n, bound); return x;

template <BidirectionalIterator I>
  I prev(I x, difference_type_t<I> n = 1);
  Effects: Equivalent to: advance(x, -n); return x;

template <BidirectionalIterator I>
  I prev(I x, difference_type_t<I> n, I bound);
  Effects: Equivalent to: advance(x, -n, bound); return x;

6.7 Iterator adaptors [iterators.predef]

6.7.1 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).

6.7.1.1 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;

      reverse_iterator();
      explicit reverse_iterator(I x);
      reverse_iterator(const reverse_iterator<ConvertibleTo<I>>& i);
      reverse_iterator& operator=(const reverse_iterator<ConvertibleTo<I>>& i);

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

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

      reverse_iterator operator+(difference_type n) const
        requires RandomAccessIterator<I>();
      reverse_iterator& operator+=(difference_type n);
\begin{verbatim}

requires RandomAccessIterator<I>();
reverse_iterator operator-(difference_type n) const
  requires RandomAccessIterator<I>();
reverse_iterator& operator-=(difference_type n)
  requires RandomAccessIterator<I>();
reference operator[] (difference_type n) const
  requires RandomAccessIterator<I>();
private:
  I current; // exposition only
};

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

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

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

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

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

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

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

  template <BidirectionalIterator I>
    reverse_iterator<I> make_reverse_iterator(I i);
\end{verbatim}

§ 6.7.1.1
6.7.1.2 reverse_iterator operations [reverse.iter.ops]

6.7.1.2.1 reverse_iterator constructor [reverse.iter.cons]

reverse_iterator();

1  Effects: Value-initializes current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type I.

explicit reverse_iterator(I x);  

2  Effects: Initializes current with x.

reverse_iterator(const reverse_iterator<ConvertibleTo<I>>& i);

3  Effects: Initializes current with i.current.

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

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

1  Effects: Assigns i.current to current.

2  Returns: *this.

6.7.1.2.3 Conversion [reverse.iter.conv]

I base() const;

1  Returns: current.

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

reference operator*() const;

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

6.7.1.2.5 operator-> [reverse.iter.opref]

pointer operator->() const;

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

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

reverse_iterator& operator++();

1  Effects: --current;

2  Returns: *this.

reverse_iterator operator++(int);

3  Effects:

    reverse_iterator tmp = *this;
    --current;
    return tmp;
6.7.1.2.7 operator--
reverse_iterator& operator--();

    Effects: ++current
    Returns: *this.

reverse_iterator operator--(int);

    Effects:
    reverse_iterator tmp = *this;
       ++current;
    return tmp;

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

    Returns: reverse_iterator(current-n).

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

    Effects: current -= n;
    Returns: *this.

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

    Returns: reverse_iterator(current+n).

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

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

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

    Returns: current[-n-1].

§ 6.7.1.2.12
6.7.1.2.13  
operator==

```cpp
template <class I1, class I2>
   requires EqualityComparable<I1, I2>()
   bool operator==(  
       const reverse_iterator<I1>& x,  
       const reverse_iterator<I2>& y);
1
   Effects: Equivalent to: return x.current == y.current;
```

6.7.1.2.14  
operator!=

```cpp
template <class I1, class I2>
   requires EqualityComparable<I1, I2>()
   bool operator!=(  
       const reverse_iterator<I1>& x,  
       const reverse_iterator<I2>& y);
1
   Effects: Equivalent to: return x.current != y.current;
```

6.7.1.2.15  
operator<

```cpp
template <class I1, class I2>
   requires StrictTotallyOrdered<I1, I2>()
   bool operator<(  
       const reverse_iterator<I1>& x,  
       const reverse_iterator<I2>& y);
1
   Effects: Equivalent to: return x.current > y.current;
```

6.7.1.2.16  
operator>

```cpp
template <class I1, class I2>
   requires StrictTotallyOrdered<I1, I2>()
   bool operator>(  
       const reverse_iterator<I1>& x,  
       const reverse_iterator<I2>& y);
1
   Effects: Equivalent to: return x.current < y.current;
```

6.7.1.2.17  
operator>=

```cpp
template <class I1, class I2>
   requires StrictTotallyOrdered<I1, I2>()
   bool operator>=  
       const reverse_iterator<I1>& x,  
       const reverse_iterator<I2>& y);
1
   Effects: Equivalent to: return x.current <= y.current;
```

6.7.1.2.18  
operator<=

```cpp
template <class I1, class I2>
   requires StrictTotallyOrdered<I1, I2>()
   bool operator<=  
       const reverse_iterator<I1>& x,  
       const reverse_iterator<I2>& y);
1
   Effects: Equivalent to: return x.current >= y.current;
```

§ 6.7.1.2.18
6.7.1.2.19  operator-  [reverse.iter.opdiff]

```
template <class I1, class I2>
  requires SizedSentinel<I1, I2>()
  difference_type_t<I2> operator-(
    const reverse_iterator<I1>& x,
    const reverse_iterator<I2>& y);
```

1  Effects: Equivalent to: return y.current - x.current;

6.7.1.2.20  operator+  [reverse.iter.opsum]

```
template <RandomAccessIterator I>
  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);

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

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

1  Returns: reverse_iterator<I>(i).

6.7.2  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,

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

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

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.

6.7.2.1  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);

    § 6.7.2.1
  }
}}}
```
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);
}}}

6.7.2.2 back_insert_iterator operations

6.7.2.2.1 back_insert_iterator constructor

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

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

6.7.2.2.2 back_insert_iterator::operator=

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.

6.7.2.2.3 back_insert_iterator::operator*

back_insert_iterator& operator*();
1 Returns: *this.

6.7.2.2.4 back_insert_iterator::operator++

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

6.7.2.2.5 back_inserter

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

§ 6.7.2.2.5
6.7.2.3 Class template 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);
}}}}

6.7.2.4 front_insert_iterator operations

6.7.2.4.1 front_insert_iterator constructor

constexpr front_insert_iterator();
1
   Effects: Value-initializes container.

explicit front_insert_iterator(Container& x);
2
   Effects: Initializes container with std::addressof(x).

6.7.2.4.2 front_insert_iterator::operator=

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.

6.7.2.4.3 front_insert_iterator::operator*

front_insert_iterator& operator*();
1
   Returns: *this.

§ 6.7.2.4.3
6.7.2.4.4 front_insert_iterator::operator++  

front_insert_iterator& operator++();
front_insert_iterator operator++(int);

1

Returns: *this.

6.7.2.4.5 front_inserter

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

1

Returns: front_insert_iterator<Container>(x).

6.7.2.5 Class template 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);
}}}}

6.7.2.6 insert_iterator operations

6.7.2.6.1 insert_iterator constructor

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 std::addressof(x) and iter with i.

6.7.2.6.2 insert_iterator::operator=

insert_iterator&
  operator=(const value_type_t<Container>& value);
Effects: Equivalent to:
    iter = container->insert(iter, value);
    ++iter;

Returns: *this.

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

Effects: Equivalent to:
    iter = container->insert(iter, std::move(value));
    ++iter;

Returns: *this.

6.7.2.6.3 insert_iterator::operator*

insert_iterator& operator*();

Returns: *this.

6.7.2.6.4 insert_iterator::operator++

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

Returns: *this.

6.7.2.6.5 inserter

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

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

6.7.3 Move iterators and sentinels

6.7.3.1 Class template move_iterator

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.

Example:

```cpp
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>;

§ 6.7.3.1
using value_type = value_type_t<I>;
using iterator_category = input_iterator_tag;
using reference = see below;

move_iterator();
explicit move_iterator(I i);
move_iterator(const move_iterator<ConvertibleTo<I>>& i);
move_iterator& operator=(const move_iterator<ConvertibleTo<I>>& i);

I base() const;
reference operator*() const;
move_iterator& operator++();
move_iterator operator++(int);
move_iterator& operator--()
    requires BidirectionalIterator<I>();
move_iterator operator--(int)
    requires BidirectionalIterator<I>();

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

private:
    I current; // exposition only
};

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

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

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

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

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

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


requires StrictTotallyOrdered<I1, I2>()
bool operator>=(
    const move_iterator<I1>& x, const move_iterator<I2>& y);

template <class I1, class I2>
    requires SizedSentinel<I1, I2>()
    difference_type_t<I2> operator-(
        const move_iterator<I1>& x,
        const move_iterator<I2>& y);

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

template <InputIterator I>
    move_iterator<I>
    operator+(difference_type_t<I> n, const move_iterator<I>& x);

}}}

move_iterator<I> make_move_iterator(I i);
}}

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

4 [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]

6.7.3.2 move_iterator operations [move.iter.ops]
6.7.3.2.1 move_iterator constructors [move.iter.op.const]

move_iterator();

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

explicit move_iterator(I i);

    Effects: Constructs a move_iterator, initializing current with i.

move_iterator(const move_iterator<ConvertibleTo<I>>& i);

    Effects: Constructs a move_iterator, initializing current with i.current.

6.7.3.2.2 move_iterator::operator= [move.iter.op=]

move_iterator& operator=(const move_iterator<ConvertibleTo<I>>& i);

    Effects: Assigns i.current to current.

6.7.3.2.3 move_iterator conversion [move.iter.op.conv]

I base() const;

    Returns: current.

6.7.3.2.4 move_iterator::operator* [move.iter.op.star]

reference operator*() const;

    Effects: Equivalent to: return static_cast<reference>(*current);

§ 6.7.3.2.4
6.7.3.2.5 move_iterator::operator++

move_iterator& operator++();

Effects: Equivalent to ++current.

Returns: *this.

move_iterator operator++(int);

Effects: Equivalent to:

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

6.7.3.2.6 move_iterator::operator--

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

Effects: Equivalent to --current.

Returns: *this.

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

Effects: Equivalent to:

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

6.7.3.2.7 move_iterator::operator+

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

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

6.7.3.2.8 move_iterator::operator+=

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

Effects: Equivalent to current += n.

Returns: *this.

6.7.3.2.9 move_iterator::operator-

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

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

6.7.3.2.10 move_iterator::operator=*

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

Effects: Equivalent to current -= n.

Returns: *this.
6.7.3.2.11 move_iterator::operator[]

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

Effects: Equivalent to: return static_cast<reference>(current[n]);

6.7.3.2.12 move_iterator comparisons

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

Effects: Equivalent to: return x.current == y.current;

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

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

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

Effects: Equivalent to: return x.current < y.current;

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

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

6.7.3.2.13 move_iterator non-member functions

template <class I1, class I2>
  requires SizedSentinel<I1, I2>()
difference_type_t<I2> operator-(
  const move_iterator<I1>& x,
  const move_iterator<I2>& y);

Effects: Equivalent to: return x.current - y.current;
template <RandomAccessIterator I>
    move_iterator<I>
    operator+(  
        difference_type_t<I> n,  
        const move_iterator<I>& x);  

Effects: Equivalent to: return x + n;

template <InputIterator I>
    move_iterator<I> make_move_iterator(I i);

Returns: move_iterator<I>(i).

6.7.3.3 Class template move_sentinel

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

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

    template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O,  
          IndirectPredicate<I> Pred>  
    requires IndirectlyMovable<ID, 0>()  
    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>
        bool operator==(  
            const move_iterator<I>& i, const move_sentinel<S>& s);  

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

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

§ 6.7.3.3
const move_sentinel<S>& s, const move_iterator<I>& i);

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

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

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

6.7.3.4 move_sentinel operations

6.7.3.4.1 move_sentinel constructors

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.

6.7.3.4.2 move_sentinel::operator=

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

1 Effects: Assigns s.last to last.

Returns: *this.

6.7.3.4.3 move_sentinel comparisons

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

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

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

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

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

2 Effects: Equivalent to: return !(i == s);
6.7.3.4.4 move_sentinel non-member functions

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

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

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

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

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

Returns: move_sentinel<S>(s).

6.7.4 Common iterators

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

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

Example:

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

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

— end example]

6.7.4.1 Class template common_iterator

namespace std { namespace experimental { namespace ranges { inline namespace v1 {  
    template <Iterator I, Sentinel<I> S>  
        requires !Same<I, S>()  
    class common_iterator {  
        public:  
            using difference_type = difference_type_t<I>;  

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

§ 6.7.4.1 84
-common_iterator();

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

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

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

} // namespace

}}} // namespace

Note: It is unspecified whether common_iterator’s members iter and sentinel have distinct addresses or not. — end note]

6.7.4.2 common_iterator operations [common.iter.ops]
6.7.4.2.1 common_iterator constructors [common.iter.op.const]

common_iterator();

Effects: Constructs a common_iterator, value-initializing is_sentinel and iter. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are
defined on a value-initialized iterator of type I.

Remarks: It is unspecified whether any initialization is performed for sentinel.

```
common_iterator(I i);
```
2
**Effects:** Constructs a common_iterator, initializing is_sentinel with false and iter with i.

```
common_iterator(S s);
```
3
**Effects:** Constructs a common_iterator, initializing is_sentinel with true and sentinel with s.

```
common_iterator(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
```
4
**Effects:** Constructs a common_iterator, initializing is_sentinel with u.is_sentinel.

(7.1) — If u.is_sentinel is true, sentinel is initialized with u.sentinel.
(7.2) — If u.is_sentinel is false, iter is initialized with u.iter.

```
common_iterator& operator=(const common_iterator<ConvertibleTo<I>, ConvertibleTo<S>>& u);
```
5
**Effects:** Assigns u.is_sentinel to is_sentinel.

(1.1) — If u.is_sentinel is true, assigns u.sentinel to sentinel.
(1.2) — If u.is_sentinel is false, assigns u.iter to iter.

```
~common_iterator();
```
6
**Effects:** Destroys all members that are currently initialized.

```
decltype(auto) operator*();
decltype(auto) operator*() const;
```
7
**Requires:** !is_sentinel

(1) **Effects:** Equivalent to: return *iter;

```
see below operator->() const requires Readable<I>();
```
8
**Requires:** !is_sentinel

(1) **Effects:** Given an object i of type I
— if I is a pointer type or if the expression \( i.\text{operator->}() \) is well-formed, this function returns \( \text{iter} \).

— Otherwise, if the expression \(*\text{iter}\) is a glvalue, this function is equivalent to return \( \text{addressof(*iter)} \);

— Otherwise, this function returns a proxy object of an unspecified type equivalent to the following:

```cpp
class proxy {
  // exposition only
  value_type_t<I> keep_; 
  proxy(reference_t<I>&& x)
    : keep_(std::move(x)) {} 
  public:
    const value_type_t<I>& operator->() const {
      return addressof(keep_); 
    }
};
```

that is initialized with \(*\text{iter}\).

### 6.7.4.2.4 common_iterator::operator++

```cpp
common_iterator& operator++();
1  Requires: !is_sentinel 
2  Effects: ++\text{iter}. 
3  Returns: *\text{this}. 
```

### 6.7.4.2.5 common_iterator comparisons

```cpp
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); 
```

```cpp
template <class I1, class I2, Sentinel<I2> S1, Sentinel<I1> S2>
requires EqualityComparable<I1, I2>()
bool operator==( 
  const common_iterator<I1, S1>& x, const common_iterator<I2, S2>& y);
2  Effects: Equivalent to:
```
return x.is_sentinel ?
    (y.is_sentinel || y.iter == x.sentinel) :
    (y.is_sentinel ?
        x.iter == y.sentinel :
        x.iter == y.iter);

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

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

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

Effects: Equivalent to:
    return x.is_sentinel ?
        (y.is_sentinel ? 0 : x.sentinel - y.iter) :
        (y.is_sentinel ?
            x.iter - y.sentinel :
            x.iter - y.iter);

6.7.5 Default sentinels

6.7.5.1 Class default_sentinel

Class default_sentinel is an empty type used to denote the end of a range. It is intended to be used
with iterator types that know the bound of their range (e.g., counted_iterator (6.7.6.1)).

6.7.6 Counted iterators

6.7.6.1 Class template counted_iterator

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

[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]

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>;

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

I base() const;
difference_type_t<I> count() const;
see below operator*();
see below operator*() const;
counted_iterator& operator++();
counted_iterator operator++(int);
counted_iterator& operator--()
  requires BidirectionalIterator<I>();
counted_iterator operator--(int)
  requires BidirectionalIterator<I>();

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

private:
  I current; // exposition only
  difference_type_t<I> cnt; // exposition only
};

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

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

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

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

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

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

difference_type_t<I> operator-(
  count_iterator<I>& n, const counted_iterator<I>& x);

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

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

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

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

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

6.7.6.2 counted_iterator operations
6.7.6.2.1 counted_iterator constructors

counted_iterator();

Effects: Constructs a counted_iterator, value-initializing current and cnt. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type I.
counted_iterator(I i, difference_type_t<I> n);

Requires: n >= 0
Effects: Constructs a counted_iterator, initializing current with i and cnt with n.

counted_iterator(const counted_iterator<ConvertibleTo<I>>& i);

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

6.7.6.2.2 counted_iterator::operator=
counted_iterator& operator=(const counted_iterator<ConvertibleTo<I>>& i);

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

6.7.6.2.3 counted_iterator conversion
I base() const;

Returns: current.

6.7.6.2.4 counted_iterator count
difference_type_t<I> count() const;

Returns: cnt.

6.7.6.2.5 counted_iterator::operator*
decltype(auto) operator*();
decltype(auto) operator*() const;

Effects: Equivalent to: return *current;

6.7.6.2.6 counted_iterator::operator++
counted_iterator& operator++();

Requires: cnt > 0
Effects: Equivalent to:
  ++current;
  --cnt;

Returns: *this.

counted_iterator operator++(int);

Requires: cnt > 0
Effects: Equivalent to:
  counted_iterator tmp = *this;
  ++current;
  --cnt;
  return tmp;
6.7.6.2.7 counted_iterator::operator--

    counted_iterator& operator--();
    requires BidirectionalIterator<I>()
    
1  Effects: Equivalent to:
    --current;
    ++cnt;

2  Returns: *this.

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

3  Effects: Equivalent to:
    counted_iterator tmp = *this;
    --current;
    ++cnt;
    return tmp;

6.7.6.2.8 counted_iterator::operator+

    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);

6.7.6.2.9 counted_iterator::operator+=

    counted_iterator& operator+=(difference_type n)
    requires RandomAccessIterator<I>();
    
1  Requires: n <= cnt

2  Effects:
    current += n;
    cnt -= n;

3  Returns: *this.

6.7.6.2.10 counted_iterator::operator-

    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);

6.7.6.2.11 counted_iterator::operator-=

    counted_iterator& operator-=(difference_type n)
    requires RandomAccessIterator<I>();
    
1  Requires: -n <= cnt

2  Effects:

§ 6.7.6.2.11
current -= n;
cnt += n;

_**Returns:**_ *this.

### 6.7.6.2.12 counted_iterator::operator[]

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

_**Requires:**_ \( n \leq \text{cnt} \)

_**Effects:**_ Equivalent to: return current[n];

### 6.7.6.2.13 counted_iterator comparisons

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

_**Requires:**_ \( x \) and \( y \) shall refer to elements of the same sequence (6.7.6).

_**Effects:**_ Equivalent to: return \( x.\text{cnt} == y.\text{cnt} \);

bool operator==(const counted_iterator<auto>& x, default_sentinel);
bool operator==(default_sentinel, const counted_iterator<auto>& x);

_**Effects:**_ Equivalent to: return \( x.\text{cnt} == 0 \);

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

_**Requires:**_ For the first overload, \( x \) and \( y \) shall refer to elements of the same sequence (6.7.6).

_**Effects:**_ Equivalent to: return \( ! (x == y) \);

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

_**Requires:**_ \( x \) and \( y \) shall refer to elements of the same sequence (6.7.6).

_**Effects:**_ Equivalent to: return \( y.\text{cnt} < x.\text{cnt} \);

_**Note:**_ The argument order in the _Effects_ element is reversed because \( \text{cnt} \) counts down, not up. — end note

template <class I1, class I2>
requires Common<I1, I2>()
bool operator<=(const counted_iterator<I1>& x, const counted_iterator<I2>& y);
Requires: x and y shall refer to elements of the same sequence (6.7.6).
Effects: Equivalent to: return !(y < x);

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

Requires: x and y shall refer to elements of the same sequence (6.7.6).
Effects: Equivalent to: return y < x;

6.7.6.2.14  counted_iterator non-member functions

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

Requires: x and y shall refer to elements of the same sequence (6.7.6).
Effects: Equivalent to: return y.cnt - x.cnt;

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

Effects: Equivalent to: return -x.cnt;

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

Effects: Equivalent to: return y.cnt;

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

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

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

Requires: n >= 0.
Returns: counted_iterator<I>(i, n).

template <Iterator I>
void advance(counted_iterator<I>& i, difference_type_t<I> n);
Requires: n <= i.cnt.

Effects:

\[ i = \text{make\_counted\_iterator}(\text{next}(i.\text{current}, n), i.\text{cnt} - n); \]

### 6.7.7 Dangling wrapper

#### 6.7.7.1 Class template dangling

Class template `dangling` is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.

```cpp
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  template <CopyConstructible T>
  class dangling {
    public:
      dangling() requires DefaultConstructible<T>();
      dangling(T t);
      T get_unsafe() const;
    private:
      T value; // exposition only
    };
    template <Range R>
    using safe_iterator_t = conditional_t<is_lvalue_reference<R>::value,
                                          iterator_t<R>,
                                          dangling<iterator_t<R>>;}
}}}
```

#### 6.7.7.2 dangling operations

##### 6.7.7.2.1 dangling constructors

\[ \text{dangling}() \text{ requires DefaultConstructible}<T>(); \]

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

\[ \text{dangling}(T t); \]

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

#### 6.7.7.2.2 dangling::get_unsafe

\[ \text{T get_unsafe()} \text{ const}; \]

1. **Returns:** `value`.

### 6.7.8 Unreachable sentinel

#### 6.7.8.1 Class unreachable

Class `unreachable` is a sentinel type that can be used with any `Iterator` to denote an infinite range. Comparing an iterator for equality with an object of type `unreachable` always returns `false`.

[Example:

```cpp
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 \texttt{unreachable} above potentially makes the call to \texttt{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;
}}}

6.7.8.2 unreachable operations

6.7.8.2.1 operator==

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

1 \textit{Returns: false.}

6.7.8.2.2 operator!

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

1 \textit{Returns: true.}

6.8 Stream iterators

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[Example:

\begin{verbatim}
partial_sum(istream_iterator<double, char>(cin),
           istream_iterator<double, char>(),
           ostream_iterator<double, char>(cout, "\n"));
\end{verbatim}

reads a file containing floating point numbers from \texttt{cin}, and prints the partial sums onto \texttt{cout}. —end example]

6.8.1 Class template \texttt{istream_iterator}

The class template \texttt{istream_iterator} is an input iterator (6.2.9) that reads (using \texttt{operator\texttt{>>}}) successive elements from the input stream for which it was constructed. After it is constructed, and every time \texttt{++} is used, the iterator reads and stores a value of \texttt{T}. If the iterator fails to read and store a value of \texttt{T} (\texttt{fail()} on the stream returns \texttt{true}), the iterator becomes equal to the \textit{end-of-stream} iterator value. The constructor with no arguments \texttt{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 \texttt{operator*} on an end-of-stream
iterator is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end-of-stream iterator is not defined. For any other iterator value a `const T*` is returned. The behavior of a program that applies `operator++()` to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators.

Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```cpp
namespace std { 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;
        istream_iterator();
        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);
}}}}
```

§ 6.8.1
6.8.1.1 istream_iterator constructors and destructor

see below istream_iterator();
see below istream_iterator(default_sentinel);

1 Effects: Constructs the end-of-stream iterator. If T is a literal type, then 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.

6.8.1.2 istream_iterator operations

const T& operator*() const;

1 Returns: value.

const T* operator->() const;

2 Effects: Equivalent to: return std::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);

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);

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);

Returns: !(x == y)

6.8.2 Class template ostream_iterator  [ostream.iterator]

ostream_iterator writes (using operator<<) successive elements onto the output stream from which it
was constructed. If it was constructed with charT* as a constructor argument, this string, called a delimiter
string, is written to the stream after every T is written. It is not possible to get a value out of the output
iterator. Its only use is as an output iterator in situations like

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

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
    };
}}}}
6.8.2.1 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.

6.8.2.2 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.

6.8.3 Class template istreambuf_iterator [istreambuf.iterator]

The class template istreambuf_iterator defines an input iterator (6.2.9) that reads successive characters from the streambuf for which it was constructed. operator* provides access to the current input character, if any. [Note: operator-> may return a proxy. — end note] Each time operator++ is evaluated, the iterator advances to the next input character. If the end of stream is reached (streambuf_type::sgetc() returns traits::eof()), the iterator becomes equal to the end-of-stream iterator value. The default constructor istreambuf_iterator() and the constructor istreambuf_iterator(nullptr) both construct an end-of-stream iterator object suitable for use as an end-of-range. All specializations of istreambuf_iterator shall have a trivial copy constructor, a constexpr default constructor, and a trivial destructor.

The result of operator*() on an end-of-stream iterator is undefined. For any other iterator value a char_ type value is returned. It is impossible to assign a character via an input iterator.

namespace std { 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 basic_streambuf<charT, traits> streambuf_type;
typedef basic_istream<charT, traits> istream_type;

class proxy;  // exposition only

constexpr istreambuf_iterator() noexcept;
constexpr istreambuf_iterator(const istreambuf_iterator&) noexcept = default;
~istreambuf_iterator() = default;
istreambuf_iterator(istream_type& s) noexcept;
istreambuf_iterator(streambuf_type* s) noexcept;
istreambuf_iterator(const proxy& p) noexcept;

charT operator*() const;
pointer operator->() const;
istreambuf_iterator& 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);

6.8.3.1 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)
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.

6.8.3.2 istreambuf_iterator constructors

### constexpr istreambuf_iterator() noexcept;

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

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

#### 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;

#### Effects:
Constructs a istreambuf_iterator<> that uses the basic_streambuf<> object pointed to by the proxy object’s constructor argument p.

6.8.3.3 istreambuf_iterator::operator* 

#### charT operator*() const

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

6.8.3.4 istreambuf_iterator::operator++

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

#### Effects:
Equivalent to sbuf_->sbumpc().

#### Returns: *this.

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

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

6.8.3.5 istreambuf_iterator::equal

#### bool equal(const istreambuf_iterator& b) const;

#### Returns: true if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what streambuf object they use.

6.8.3.6 operator==

#### template <class charT, class traits>

#### bool operator==(const istreambuf_iterator<charT, traits>& a, const istreambuf_iterator<charT, traits>& b);
Effects: Equivalent to: \texttt{return a.equal(b);} \\
\begin{verbatim}
template <class charT, class traits>
  bool operator==(default_sentinel a, 
                  const istreambuf_iterator<charT, traits>& b);
\end{verbatim}

Effects: Equivalent to: \texttt{return istreambuf_iterator<charT, traits>().equal(b);} \\
\begin{verbatim}
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT, traits>& a, 
                  default_sentinel b);
\end{verbatim}

Effects: Equivalent to: \texttt{return a.equal(istreambuf_iterator<charT, traits>{});} \\

6.8.3.7 \texttt{operator!=} \hspace{1cm} \texttt{[istreambuf.iterator::op!=]} \\
\begin{verbatim}
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT, traits>& a, 
                  const istreambuf_iterator<charT, traits>& b);
\end{verbatim}

Effects: Equivalent to: \texttt{return !(a == b);} \\

6.8.4 Class template \texttt{ostreambuf_iterator} \hspace{1cm} \texttt{[ostreambuf.iterator]} \\
\begin{verbatim}
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
    };
  }}}
\end{verbatim}

The class template \texttt{ostreambuf_iterator} writes successive \textit{characters} onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

6.8.4.1 \texttt{ostreambuf_iterator} constructors \hspace{1cm} \texttt{[ostreambuf.iter.cons]}

§ 6.8.4.1
constexpr ostreambuf_iterator() noexcept;

Effects: Initializes sbuf_ with nullptr.

ostreambuf_iterator(ostream_type& s) noexcept;

Requires: s.rdbuf() != nullptr.

Effects: Initializes sbuf_ with s.rdbuf().

ostreambuf_iterator(streambuf_type* s) noexcept;

Requires: s != nullptr.

Effects: Initializes sbuf_ with s.

6.8.4.2 ostreambuf_iterator operations

ostreambuf_iterator&
operator=(charT c);

Requires: sbuf_ != nullptr.

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

Returns: *this.

ostreambuf_iterator& operator*();

Returns: *this.

ostreambuf_iterator& operator++();
ostreambuf_iterator operator++(int);

Returns: *this.

bool failed() const noexcept;

Requires: sbuf_ != nullptr.

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

6.9 Range concepts

6.9.1 General

This subclause describes components for dealing with ranges of elements.

The following subclauses describe range and view requirements, and components for range primitives, pre-defined ranges, and stream ranges, as summarized in Table 7.

Table 7 — Ranges library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.9.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>6.10</td>
<td>Range access</td>
</tr>
<tr>
<td>6.11</td>
<td>Range primitives</td>
</tr>
</tbody>
</table>

<experimental/ranges/iterator>
6.9.2 Range requirements

6.9.2.1 In general

Ranges are an abstraction of containers that allow a C++ program to operate on elements of data structures uniformly. In their simplest form, a range object is one on which one can call `begin` and `end` to get an iterator (6.2.6) and a sentinel (6.2.7). To be able to construct template algorithms and range adaptors that work correctly and efficiently on different types of sequences, the library formalizes not just the interfaces but also the semantics and complexity assumptions of ranges.

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 8.

Table 8 — Relations among range categories

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

The `Range` concept requires only that `begin` and `end` return an iterator and a sentinel. The `SizedRange` concept refines `Range` with the requirement that the number of elements in the range can be determined in constant time using the `size` function. The `View` concept specifies requirements on an `Range` type with constant-time copy and assign operations.

In addition to the three fundamental range categories, this document defines a number of convenience refinements of `Range` that group together requirements that appear often in the concepts, algorithms, and range adaptors. `Bounded ranges` are ranges for which `begin` and `end` return objects of the same type. `Random access ranges` are ranges for which `begin` returns a type that satisfies `RandomAccessIterator` (6.2.13). The range categories `bidirectional ranges`, `forward ranges`, `input ranges`, and `output ranges` are defined similarly.

6.9.2.2 Ranges

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

```
template <class T>
using iterator_t = decltype(ranges::begin(declval<T&>()));
```

```
template <class T>
using sentinel_t = decltype(ranges::end(declval<T&>()));
```

```
template <class T>
concept bool Range() {
    return requires(T& t) {
        ranges::end(t);
    };
}
```

Given an lvalue `t` of type `remove_reference_t<T>&`, `Range<T>()` is satisfied if and only if

1. `begin(t), end(t)` denotes a range.
2. Both `begin(t)` and `end(t)` are amortized constant time and non-modifying. 
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]

6.9.2.3 Sized ranges [ranges.sized]
1 The SizedRange concept specifies the requirements of a Range type that knows its size in constant time with the size function.

```cpp
template <class>
constexpr bool disable_sized_range = false;

template <class T>
concept bool SizedRange() {
  return Range<T>() &&
  !disable_sized_range<remove_cv_t<remove_reference_t<T>>> &&
  requires(const remove_reference_t<T>& t) {
    { ranges::size(t) } -> ConvertibleTo<difference_type_t<iterator_t<T>>; }
};
```

2 Given an lvalue t of type remove_reference_t<T>, SizedRange<T>() is satisfied if and only if:
   (2.1) — size(t) returns the number of elements in t.
   (2.2) — If iterator_t<T> satisfies ForwardIterator, size(t) is well-defined regardless of the evaluation of begin(t). [Note: size(t) is otherwise not required be well-defined after evaluating begin(t). For a SizedRange whose iterator type does not model ForwardIterator, for example, size(t) might only be well-defined if evaluated before the first call to begin(t). —end note]

3 [Note: The disable_sized_range predicate provides a mechanism to enable use of range types with the library that meet the syntactic requirements but do not in fact satisfy SizedRange. A program that instantiates a library template that requires a Range with such a range type R is ill-formed with no diagnostic required unless disable_sized_range<remove_cv_t<remove_reference_t<R>>> evaluates to true (3.3.1.3). —end note]

6.9.2.4 Views [ranges.view]
1 The View concept specifies the requirements of a Range type that has constant time copy, move and assignment operators; that is, the cost of these operations is not proportional to the number of elements in the View.

2 [Example: Examples of Views are:
   (2.1) — A Range type that wraps a pair of iterators.
   (2.2) — A Range type that holds its elements by shared_ptr and shares ownership with all its copies.
   (2.3) — A Range type that generates its elements on demand.
   A container (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]

```cpp
template <class T>
struct enable_view { };

struct view_base { };
```
template <class T>
constexpr bool __view_predicate = see below;

template <class T>
concept bool View() {
    return Range<T>() &&
    Semiregular<T>() &&
    __view_predicate<T>;
}

Since the difference between Range and View is largely semantic, the two are differentiated with the help of the enable_view trait. Users may specialize enable_view to derive from true_type or false_type.

For a type T, the value of __view_predicate<T> shall be:

- If enable_view<T> has a member type type, enable_view<T>::type::value;
- Otherwise, if T is derived from view_base, true;
- Otherwise, if both T and const T satisfy Range and reference_t<iterator_t<const 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]
- Otherwise, true.

6.9.2.5 Bounded ranges [ranges.bounded]

The BoundedRange concept specifies requirements of an 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]

```cpp
template <class T>
concept bool BoundedRange() {
    return Range<T>() && Same<iterator_t<T>, sentinel_t<T>>();
}
```

6.9.2.6 Input ranges [ranges.input]

The InputRange concept specifies requirements of an Range type for which begin returns a type that satisfies InputIterator (6.2.9).

```cpp
template <class T>
concept bool InputRange() {
    return Range<T>() && InputIterator<iterator_t<T>>();
}
```

6.9.2.7 Output ranges [ranges.output]

The OutputRange concept specifies requirements of an Range type for which begin returns a type that satisfies OutputIterator (6.2.10).

```cpp
template <class R, class T>
concept bool OutputRange() {
    return Range<R>() && OutputIterator<iterator_t<R>, T>();
}
```
6.9.2.8 Forward ranges

1 The `ForwardRange` concept specifies requirements of an `InputRange` type for which `begin` returns a type that satisfies `ForwardIterator` (6.2.11).

```cpp
template <class T>
concept bool ForwardRange() {
    return InputRange<T>() && ForwardIterator<iterator_t<T>>();
}
```

6.9.2.9 Bidirectional ranges

1 The `BidirectionalRange` concept specifies requirements of a `ForwardRange` type for which `begin` returns a type that satisfies `BidirectionalIterator` (6.2.12).

```cpp
template <class T>
concept bool BidirectionalRange() {
    return ForwardRange<T>() && BidirectionalIterator<iterator_t<T>>();
}
```

6.9.2.10 Random access ranges

1 The `RandomAccessRange` concept specifies requirements of a `BidirectionalRange` type for which `begin` returns a type that satisfies `RandomAccessIterator` (6.2.13).

```cpp
template <class T>
concept bool RandomAccessRange() {
    return BidirectionalRange<T>() && RandomAccessIterator<iterator_t<T>>();
}
```

6.10 Range access

6.10.1 `begin`

1 The name `begin` denotes a customization point object (3.3.2.3). The effect of the expression `ranges::begin(E)` for some expression `E` is 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.

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

The name end denotes a customization point object (3.3.2.3). The effect of the expression ranges::end(E) for some expression E is equivalent to:

1. 
2. 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]

2. Otherwise, (E) + extent<T>::value if E has array type (ISO/IEC 14882:2014 §3.9.2) T.

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.

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.

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

6.10.3 cbegin

The name cbegin denotes a customization point object (3.3.2.3). The effect of the expression ranges::cbegin(E) for some expression E of type T is equivalent to ranges::begin(static_const<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]

6.10.4 cend

The name cend denotes a customization point object (3.3.2.3). The effect of the expression ranges::cend(E) for some expression E of type T is 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]

6.10.5 rbegin

The name rbegin denotes a customization point object (3.3.2.3). The effect of the expression ranges::rbegin(E) for some expression E is equivalent to:

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]

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>() (6.2.12).

(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 ]

6.10.6 rend

The name rend denotes a customization point object (3.3.2.3). The effect of the expression ranges::rend(E)
for some expression E is 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>() (6.2.12).

(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 ]

6.10.7 crbegin

The name crbegin denotes a customization point object (3.3.2.3). The effect of the expression ranges::crbegin(E)
for some expression E of type T is 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 ]

6.10.8 crend

The name crend denotes a customization point object (3.3.2.3). The effect of the expression ranges::crend(E)
for some expression E of type T is 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 ]

6.11 Range primitives

Template <Range R>
difference_type_t<iterator_t<R>> distance(R&& r);

1 Effects: Equivalent to: return ranges::distance(ranges::begin(r), ranges::end(r));

template <SizedRange R>
difference_type_t<iterator_t<R>> distance(R&& r);

2 Effects: Equivalent to: return ranges::size(r);
6.11.1 size

The name size denotes a customization point object (3.3.2.3). The effect of the expression `ranges::size(E)` for some expression `E` with type `T` is equivalent to:

1. If `T` is an array type, `extent<T>::value` (ISO/IEC 14882:2014 §3.9.2).
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>` (6.9.2.3) is false.
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.
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 of `ranges::begin(E)` and `ranges::end(E)` meet the syntactic requirements of `SizedSentinel<S, I>()` (6.2.8) and `ForwardIterator<I>()`. If `SizedSentinel` and `ForwardIterator` are not satisfied, the program is ill-formed with no diagnostic required.
5. Otherwise, `ranges::size(E)` is ill-formed.

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

6.11.2 empty

The name empty denotes a customization point object (3.3.2.3). The effect of the expression `ranges::empty(E)` for some expression `E` is equivalent to:

1. If `E` is a valid expression, `bool(E).empty()`.
2. Otherwise, `ranges::size(E) == 0` if it is a valid expression.
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`.
4. Otherwise, `ranges::empty(E)` is ill-formed.

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

6.11.3 data

The name data denotes a customization point object (3.3.2.3). The effect of the expression `ranges::data(E)` for some expression `E` is equivalent to:

1. 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]

2. Otherwise, `DECAY_COPY((E).data())` if it is a valid expression of pointer to object type.
3. Otherwise, `ranges::begin(E)` if it is a valid expression of pointer to object type.
4. Otherwise, `ranges::data(E)` is ill-formed.

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

The name `cdata` denotes a customization point object (3.3.2.3). The effect of the expression `ranges::cdata(E)` for some expression `E` of type `T` is equivalent to `ranges::data(static_cast<const T&>(E))`.

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

[Note: Whenever `ranges::cdata(E)` is a valid expression, it has pointer to object type. — end note]
7 Algorithms library

7.1 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, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 9.

Table 9 — Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3 Non-modifying sequence operations</td>
<td>&lt;experimental/ranges/algorithm&gt;</td>
</tr>
<tr>
<td>7.4 Mutating sequence operations</td>
<td></td>
</tr>
<tr>
<td>7.5 Sorting and related operations</td>
<td></td>
</tr>
<tr>
<td>7.6 C library algorithms</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

Header <experimental/ranges/algorithm> synopsis

#include <initializer_list>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
namespace tag {
    // 7.2, tag specifiers (See 5.5.2):
    struct in;
    struct in1;
    struct in2;
    struct out;
    struct out1;
    struct out2;
    struct fun;
    struct min;
    struct max;
    struct begin;
    struct end;
}

    // 7.3, non-modifying sequence operations:
    template <InputIterator I, Sentinel<I> S, class Proj = identity,
               IndirectPredicate<projected<I, Proj>> Pred>
    bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
           IndirectPredicate<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,
           IndirectPredicate<projected<I, Proj>> Pred>
    bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
IndirectPredicate<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,
IndirectPredicate<projected<I, Proj>> Pred>
bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
IndirectPredicate<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,
IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
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,
IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
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,
IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
safe_iterator_t<Rng>
    find_if(I first, S last, Pred pred, Proj proj = Proj{});

template <InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
safe_iterator_t<Rng>
    find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
safe_iterator_t<Rng>
    find_end(I1 first1, S1 last1, I2 first2, S2 last2,
    Pred pred = Pred{}, Proj proj = Proj{});

§ 7.1 114
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, IndirectPredicate<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, IndirectPredicate<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 <InputRange Rng, class T, class Proj = identity>
requires IndirectRelation<equal_to<>, projected<iterator_t<Rng>, 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, IndirectPredicate<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, IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
difference_type_t<iterator_t<Rng>>
count_if(Rng&& rng, Pred pred, Proj proj = Proj{});

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, class Proj1 = identity, class Proj2 = identity, IndirectPredicate<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
tagged_pair<tag::in1(I1), tag::in2(I2)>
mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <InputRange Rng1, InputIterator I2, class Proj1 = identity, class Proj2 = identity, IndirectPredicate<iterator_t<Rng1>, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(I2)>
mismatch(Rng1&& rng1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, class Proj1 = identity, class Proj2 = identity, IndirectPredicate<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, IndirectPredicate<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{});

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity> requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool equal(I1 first1, S1 last1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <InputRange Rng1, InputIterator I2, class Pred = equal_to<>, class Proj1 = identity> requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool equal(Rng1&& rng1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity> requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool equal(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
§ 7.1 116
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{});

// A.2 (deprecated):
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool is_permutation(I1 first1, S1 last1, I2 first2,
                    Pred pred = Pred{},
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <ForwardRange Rng1, ForwardIterator I2, class Pred = equal_to<>,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool is_permutation(Rng1&& rng1, I2 first2, Pred pred = Pred{},
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
                    Pred pred = Pred{},
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <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>()
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>()
bool is_permutation(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
                    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity,
class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
I
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{});

// 7.4, modifying sequence operations:
// 7.4.1, 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,
IndirectPredicate<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,
IndirectPredicate<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);

// 7.4.2, 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);

// 7.4.3, swap, A.2 (deprecated):
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in1(I1), tag::in2(I2)>
swap_ranges(I1 first1, S1 last1, I2 first2);

// A.2 (deprecated):
template <ForwardRange Rng, ForwardIterator I>
requires IndirectlySwappable<iterator_t<Rng>, I>()
tagged_pair<tag::in1(safe_iterator_t<Rng>), tag::in2(I)>
swap_ranges(Rng&& rng1, I first2);

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in(I1), tag::in2(I2)>
swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template <ForwardRange Rng1, ForwardRange Rng2>
requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>()
tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
swap_ranges(Rng1&& rng1, Rng2&& rng2);

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class F, class Proj = identity>
requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>()
tagged_pair<tag::in(I), tag::out(O)>
transform(I first, S last, O result, F op, Proj proj = Proj());

template <InputRange Rng, WeaklyIncrementable O, class F, class Proj = identity>
requires Writable<O, indirect_result_of_t<F&((projected<iterator_t<Rng>, Proj))>>()
tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
transform(Rng&& rng, O result, F op, Proj proj = Proj());

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, WeaklyIncrementable O, class F, class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_of_t<F&((projected<I2, Proj2>))>()
tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
transform(I1 first1, S1 last1, I2 first2, O result,
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{};

// A.2 (deprecated):
template <InputRange Rng, InputIterator I, WeaklyIncrementable O, class F,
    class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_of_t<F&(
        projected<iterator_t<Rng>, Proj1>, projected<I, Proj2>)>>(
    tagged_tuple<tag::in1(safe_iterator_t<Rng>), tag::in2(I), tag::out(O))
    transform(Rng&& rng1, I first2, O result,
        F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class F, class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_of_t<F&(projected<I1, Proj1>,
        projected<I2, Proj2>)>>(
    tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O))
    transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
        F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class F,
    class Proj1 = identity, class Proj2 = identity>
    requires Writable<O, indirect_result_of_t<F&(
        projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>(
    tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
        tag::in2(safe_iterator_t<Rng2>),
        tag::out(O))
    transform(Rng1&& rng1, Rng2&& rng2, O result,
        F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator 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 <ForwardRange 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 <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectPredicate<projected<I, Proj>> Pred>
    requires Writable<I, const T&>()
    I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectPredicate<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<\texttt{I}, \texttt{O}>() &&
  IndirectRelation\langle\texttt{equal_to}>, \text{projected}\langle\texttt{I}, \texttt{Proj}\rangle, \text{const} \texttt{T1}*>()\n  tagged_pair\langle\texttt{tag::in(\texttt{I})}, \text{tag::out(\texttt{O})}\rangle
  \texttt{replace_copy(I first, S last, 0 result, const} \texttt{T1& old_value, const T2& new_value,}
  \texttt{Proj proj = Proj{}});\n
\text{template \langle InputRange Rng, class T1, class T2, OutputIterator\langle const T2&\rangle O,}
\text{class Proj = identity \rangle}
\text{requires IndirectlyCopyable<\texttt{iterator\_t}\langle\texttt{Rng}\rangle, \texttt{O}()> &&
  IndirectRelation\langle\texttt{equal_to}>, \text{projected}\langle\texttt{iterator\_t}\langle\texttt{Rng}\rangle, \texttt{Proj}\rangle, \text{const} \texttt{T1}*>()\n  tagged_pair\langle\texttt{tag::in(\texttt{safe\_iterator\_t}\langle\texttt{Rng}\rangle)}, \text{tag::out(0)}\rangle
  \texttt{replace_copy(Rng\&\& rng, 0 result, const} \texttt{T1& old_value, const T2& new_value,}
  \texttt{Proj proj = Proj{}});\n
\text{template \langle InputIterator I, Sentinel\langle I\rangle S, class T, OutputIterator\langle const T&\rangle O,}
\text{class Proj = identity, IndirectPredicate\langle\texttt{projected}\langle I, \texttt{Proj}\rangle\rangle Pred \rangle}
\text{requires IndirectlyCopyable<\texttt{I}, \texttt{O}()>\n  tagged_pair\langle\texttt{tag::in(I)}, \text{tag::out(0)}\rangle
  \texttt{replace_copy_if(I first, S last, 0 result, Pred pred, const} \texttt{T& new_value,}
  \texttt{Proj proj = Proj{}});\n
\text{template \langle InputRange Rng, class T, OutputIterator\langle const T\&\rangle O,\rangle\n\text{class Proj = identity, IndirectPredicate\langle\texttt{projected}\langle\texttt{iterator\_t}\langle\texttt{Rng}\rangle, \text{Proj}\rangle\rangle Pred \rangle}
\text{requires IndirectlyCopyable<\texttt{iterator\_t}\langle\texttt{Rng}\rangle, \text{O}()>\n  tagged_pair\langle\texttt{tag::in(\texttt{safe\_iterator\_t}\langle\texttt{Rng}\rangle)}, \text{tag::out(0)}\rangle
  \texttt{replace_copy_if(Rng\&\& rng, 0 result, Pred pred, const} \texttt{T& new_value,}
  \texttt{Proj proj = Proj{}});\n
\text{template \langle class T, OutputIterator\langle const T&\rangle O, Sentinel\langle O\rangle S \rangle}
\text{O \texttt{fill}(O first, S last, const T& value);}\n
\text{template \langle class T, OutputRange\langle const T&\rangle Rng \rangle}
\text{safe\_iterator\_t\langle Rng\rangle}
\text{fill(Rng\&\& rng, const T& value);}\n
\text{template \langle class T, OutputIterator\langle const T\&\rangle O \rangle}
\text{O \texttt{fill\_n}(O first, \texttt{difference\_type\_t}\langle O\rangle n, const T& value);}\n
\text{template \langle Invocable \texttt{F}, OutputIterator\langle result\_of\_t\langle \text{F\&}()\rangle\rangle O,\rangle\n\text{Sentinel\langle O\rangle S \rangle}
\text{O \texttt{generate}(O first, S last, \texttt{F}\ gen);}\n
\text{template \langle Invocable \texttt{F}, OutputRange\langle result\_of\_t\langle \text{F\&}()\rangle\rangle Rng \rangle}
\text{safe\_iterator\_t\langle Rng\rangle}
\text{generate(Rng\&\& rng, \texttt{F}\ gen);}\n
\text{template \langle Invocable \texttt{F}, OutputIterator\langle result\_of\_t\langle \text{F\&}()\rangle\rangle O \rangle}
\text{O \texttt{generate\_n}(O first, \texttt{difference\_type\_t}\langle O\rangle n, \texttt{F}\ gen);}\n
\text{template \langle ForwardIterator I, Sentinel\langle I\rangle S, class T, class Proj = identity \rangle}
\text{requires \text{Permutable}\langle I\rangle() \&\&
  IndirectRelation\langle\texttt{equal\_to}, \text{projected}\langle I, \texttt{Proj}\rangle, \text{const} \texttt{T}*>()\n  \texttt{I remove(I first, S last, const} \texttt{T\& value, Proj proj = Proj{}});\n
\text{template \langle ForwardRange Rng, class T, class Proj = identity \rangle

\text{\S 7.1 121}
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,
    IndirectPredicate<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,
    IndirectPredicate<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>
requires IndirectlyCopyable<I, O>()
    && IndirectRelation<equal_to<>, projected<I, Proj>, const T*>()
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,
    IndirectPredicate<projected<iterator_t<Rng>, Proj>>, Pred>
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_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>() ||
ForwardIterator<O>() || IndirectlyCopyableStorable<I, O>()
tagged_pair<tag::in(I), tag::out(O)>
unique_copy(I first, S last, O result, R comp = R{}, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class Proj = identity,
  IndirectRelation<projected<iterator_t<Rng>, Proj>> R = equal_to<>>
requires IndirectlyCopyable<iterator_t<Rng>, O>() &&
  (ForwardIterator<iterator_t<Rng>>(), 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);

// 7.4.12, shuffle:
template <RandomAccessIterator I, Sentinel<I> S, class Gen>
requires Permutable<I>()
  UniformRandomNumberGenerator<remove_reference_t<Gen>>()
  __attribute__((always_inline))
  __attribute__((vector_size(16))
  __attribute__((trusted))
  __attribute__((malloc))
  __attribute__((hot))
  __attribute__((no_cold))
  __attribute__((naked))

§ 7.1
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);

// 7.4.13, partitions:
template <InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectPredicate<projected<I, Proj>> Pred>
bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
IndirectPredicate<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,
IndirectPredicate<projected<I, Proj>> Pred>
requires Permutable<I>()
I partition(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
IndirectPredicate<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,
IndirectPredicate<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,
IndirectPredicate<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, IndirectPredicate<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,
IndirectPredicate<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, 01 out_true, 02 out_false, Pred pred, Proj proj = Proj{});

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectPredicate<projected<I, Proj>> Pred>
I partition_point(I first, S last, Pred pred, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
    IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
safe_iterator_t<Rng>
    partition_point(Rng&& rng, Pred pred, Proj proj = Proj{});

// 7.5, sorting and related operations:
// 7.5.1, 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>
safe_iterator_t<Rng>
nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});

// 7.5.3, 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{});

// 7.5.4, 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{});
// 7.5.5, 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>

§ 7.1 128
requires Mergeable<iterator_t<Rng1>, iterator_t<Rng2>, 0, Comp, Proj1, Proj2>()
tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::out(O)>
    set_difference(Rng1&& rng1, Rng2&& rng2, 0 result,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable 0, class Comp = less<>, class Proj1 = identity, class Proj2 = identity
requires Mergeable<I1, I2, 0, Comp, Proj1, Proj2>()
tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
    set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, 0 result,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{},
    Proj2 proj2 = Proj2{});

// 7.5.6, 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_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{});

// 7.5.7, 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,
§ 7.1
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{});

// 7.5.9, 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. When such a version is provided
for algorithm it is called algorithm_copy. Algorithms that take predicates end with the suffix _if (which
follows the suffix _copy).

[ Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy
those function objects freely. Programmers for whom object identity is important should consider using a
wrapper class that points to a noncopied implementation object such as reference_wrapper<T> (ISO/IEC
14882:2014 §20.9.3), or some equivalent solution. —end note ]

6 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

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.

§ 7.1
return distance(a, b);

In the description of algorithm return values, sentinel values are sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator as follows:

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

Overloads of algorithms that take Range arguments (6.9.2.2) behave as if they are implemented by calling begin and end on the Range and dispatching to the overload that takes separate iterator and sentinel arguments.

Some algorithms declare both an overload that takes a Range and an Iterator, and an overload that takes two Range parameters. Since an array type (ISO/IEC 14882:2014 §3.9.2) both satisfies Range and decays to a pointer (ISO/IEC 14882:2014 §4.2) which satisfies Iterator, such overloads are ambiguous when an array is passed as the second argument. Implementations provide a mechanism to resolve this ambiguity in favor of the overload that takes two ranges.

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

Despite that the algorithm declarations nominally accept parameters by value, it is unspecified when and if the argument expressions are used to initialize the actual parameters except that any such initialization shall be sequenced before (ISO/IEC 14882:2014 §1.9) the algorithm returns. [Note: The behavior of a program that modifies the values of the actual argument expressions is consequently undefined unless the algorithm return happens before (ISO/IEC 14882:2014 §1.10) any such modifications. —end note]

### 7.2 Tag specifiers

```c
namespace tag {
    struct in { /* implementation-defined */ };
    struct in1 { /* implementation-defined */ };
    struct in2 { /* implementation-defined */ };
    struct out { /* implementation-defined */ };
    struct out1 { /* implementation-defined */ };
    struct out2 { /* implementation-defined */ };
    struct fun { /* implementation-defined */ };
    struct min { /* implementation-defined */ };
    struct max { /* implementation-defined */ };
    struct begin { /* implementation-defined */ };
    struct end { /* implementation-defined */ };
}
```

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

tag::\( X \) is a tag specifier (5.5.2) such that \( \text{TAGGET}(D, \text{tag::} X, N) \) names a tagged getter (5.5.2) with DerivedCharacteristic \( D \), ElementIndex \( N \), and ElementName \( X \).

[Example: \text{tag::in} is a type such that \( \text{TAGGET}(D, \text{tag::in}, N) \) names a type with the following interface:

```c
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)); }
};
```
7.3 Non-modifying sequence operations

7.3.1 All of

template <InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectPredicate<projected<I, Proj>> Pred>
  bool all_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
          IndirectPredicate<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.

7.3.2 Any of

template <InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectPredicate<projected<I, Proj>> Pred>
  bool any_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
          IndirectPredicate<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.

7.3.3 None of

template <InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectPredicate<projected<I, Proj>> Pred>
  bool none_of(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity,
          IndirectPredicate<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.

7.3.4 For each

template <InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectInvocable<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, 
IndirectInvocable<ProjectedIterator_t<Rng>, Proj>> Fun>
tagged_pair<tag::in(safe_iterator_t<Rng>), tag::fun(Fun)>
for_each(Rng&& rng, Fun f, Proj proj = Proj{});

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]

Returns: {last, std::move(f)}.

Complexity: Applies f and proj exactly last - first times.
Remarks: If f returns a result, the result is ignored.

7.3.5 Find

template <InputIterator I, Sentinel<I> S, class T, class Proj = identity>
requires IndirectRelation<equal_to<>, ProjectedIter<I>, 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<>, ProjectedIter<iterator_t<Rng>>, Proj>, const T*>() 
safe_iterator_t<Rng>
find(Rng&& rng, const T& value, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity, 
IndirectPredicate<ProjectedIter<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, 
IndirectPredicate<ProjectedIter<I>, Proj>> Pred>
I find_if_not(I first, S last, Pred pred, Proj proj = Proj{});

template <InputRange Rng, class Proj = identity, 
IndirectPredicate<ProjectedIter<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.

7.3.6 Find end

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2, class Proj = identity, 
IndirectRelation<I2, ProjectedIter<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{});

Effects: Finds a subsequence of equal values in a sequence.

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

Complexity: At most \((last2 - first2) \times (last1 - first1 - (last2 - first2) + 1)\) applications of the corresponding predicate and projection.

7.3.7 Find first of

template <InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectPredicate<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{});

Effects: Finds an element that matches one of a set of values.

Returns: The first iterator \(i\) in the range \([first1, last1)\) such that for some iterator \(j\) in the range \([first2, last2)\) the following condition holds: \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, *i), \text{invoke}(\text{proj2}, *j)) \neq \text{false}\). Returns \(last1\) if \([first2, last2)\) is empty or if no such iterator is found.

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

7.3.8 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{});

Effects: Finds an element that matches one of a set of values.

Returns: The first iterator \(i\) such that both \(i\) and \(i + 1\) are in the range \([first, last)\) for which the following corresponding condition holds: \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *(i + 1))) \neq \text{false}\). Returns \(last\) if no such iterator is found.
Complexity: For a nonempty range, exactly $\min((i - \text{first}) + 1, (\text{last} - \text{first}) - 1)$ applications of the corresponding predicate, where $i$ is adjacent_find's return value, and no more than twice as many applications of the projection.

7.3.9 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,
    IndirectPredicate<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,
    IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
    difference_type_t<iterator_t<Rng>>
    count_if(Rng&& rng, Pred pred, Proj proj = Proj{});

Effects: Returns the number of iterators $i$ in the range [first, last) for which the following corresponding conditions hold: $\text{invoke}(\text{proj}, *i) == \text{value}$, $\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *i)) != \text{false}$.

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

7.3.10 Mismatch

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectPredicate<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>>
    tagged_pair<tag::in1(I1), tag::in2(I2)>
    mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{},
        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <InputRange Rng1, InputIterator I2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectPredicate<projected<iterator_t<Rng1>, Proj1>,
        projected<I2, Proj2>> Pred = equal_to<>>
    tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(I2)>
    mismatch(Rng1&& rng1, I2 first2, Pred pred = Pred{},
        Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectPredicate<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,
    IndirectPredicate<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{});

Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

Returns: A pair of iterators i and j such that j == first2 + (i - first1) and i is the first iterator
in the range [first1, last1) for which the following corresponding conditions hold:

(2.1) — j is in the range [first2, last2).
(2.2) — !(*i == *(first2 + (i - first1)))
(2.3) — 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.

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

7.3.11 Equal

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool equal(I1 first1, S1 last1, I2 first2, Pred pred = Pred{},
    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <InputRange Rng1, InputIterator I2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool equal(Rng1&& rng1, I2 first2, Pred pred = Pred{},
    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
    Pred pred = Pred{},
    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Pred = equal_to<>,
    class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
bool equal(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{},
    Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if for every
iterator i in the range [first1, last1) the following condition holds: invoke(pred, invoke(proj1,
*1), invoke(proj2, *(first2 + (i - first1)))) != false. Otherwise, returns false.

§ 7.3.11 138
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.

7.3.12 Is permutation [alg.is__permutation]

// A.2 (deprecated):
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool is_permutation(I1 first1, S1 last1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <ForwardRange Rng1, ForwardIterator I2, class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool is_permutation(Rng1&& rng1, I2 first2, Pred pred = Pred{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2, class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, S1, S2, 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{});

Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

Results: 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.

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 distance(first1, last1) applications of the corresponding predicate and projections if equal(first1, last1, first2, last2, pred, proj1, proj2) would return true; otherwise, at worst $\Theta(N^2)$, where $N$ has the value distance(first1, last1).

7.3.13 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, S1, S2, Pred, Proj1, Proj2>()
I1
search(I1 first1, S1 last1, I2 first2, S2 last2,
template <ForwardRange Rng1, ForwardRange Rng2, class Pred = equal_to<>, 
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, iterator_t<Rng2>, Pred, Proj1, Proj2>()
safe_iterator_t<Rng1>
search(Rng1&& rng1, Rng2&& rng2, Pred pred = Pred{}, 
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Effects: Finds a subsequence of equal values in a sequence.

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: \( \text{invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) } \neq \text{false} \). Returns \( first1 \) if \([first2, last2)\) is empty, otherwise returns \( last1 \) if no such iterator is found.

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

template <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{});

Effects: Finds a subsequence of equal values in a sequence.

Returns: The first iterator \( i \) in the range \([first, last - count)\) such that for every non-negative integer \( n \) less than \( count \) the following condition holds: \( \text{invoke(pred, invoke(proj, *(i + n)), value) } \neq \text{false} \). Returns \( last \) if no such iterator is found.

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

7.4 Mutating sequence operations

7.4.1 Copy
template <InputIterator I, WeaklyIncrementable O>
requires IndirectlyCopyable<I, O>()
tagged_pair<tag::in(I), tag::out(O)>
copy_n(I first, difference_type_t<I> n, O result);

Effects: For each non-negative integer \( i < n \), performs \( *(result + i) = *(first + i) \).
Returns: \{first + n, result + n\}.
Complexity: Exactly \( n \) assignments.

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
IndirectPredicate<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{});

Requires: The ranges \([first, last)\) and \([result, result + (last - first))\) shall not overlap.
Effects: Copies all of the elements referred to by the iterator \( i \) in the range \([first, last)\) for which
\( \text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *i)) \) is true.
Returns: \{last, result + (last - first)\}.
Complexity: Exactly last - first applications of the corresponding predicate and projection.
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);

Effects: Copies elements in the range \([first, last)\) into the range \([result - (last-first), result)\)
starting from \( last - 1 \) and proceeding to \( first \). For each positive integer \( n <= (last - first) \),
performs \( *(result - n) = *(last - n) \).
Requires: result shall not be in the range \([first, last)\).
Returns: \{last, result - (last - first)\}.
Complexity: Exactly last - first assignments.

5) \( \text{copy_backward} \) should be used instead of \( \text{copy} \) when \( last \) is in the range \([result - (last-first), result)\).
7.4.2 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>
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 *\((result + n) = \text{std::move(*}(first + n))\).

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 *\((result - n) = \text{std::move(*}(last - n))\).

6 Requires: result shall not be in the range \((first, last]\).

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

8 Complexity: Exactly \(last - first\) assignments.

7.4.3 swap [alg.swap]

// A.8 (deprecated):
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in1(I1), tag::in2(I2)>
swap_ranges(I1 first1, S1 last1, I2 first2);

// A.2 (deprecated):
template <ForwardRange Rng, ForwardIterator I>
requires IndirectlySwappable<iterator_t<Rng>, I>()
tagged_pair<tag::in1(safe_iterator_t<Rng>), tag::in2(I)>
swap_ranges(Rng&& rngl, I first2);

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2>
requires IndirectlySwappable<I1, I2>()

6) move_backward should be used instead of move when last is in the range \([result - (last - first), result)\).

§ 7.4.3 142
tagged_pair<tag::in1(I1), tag::in2(I2)>
  swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template <ForwardRange Rng1, ForwardRange Rng2>
  requires IndirectlySwappable<iterator_t<Rng1>, iterator_t<Rng2>>()
tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(safe_iterator_t<Rng2>)>
  swap_ranges(Rng1&& rng1, Rng2&& rng2);

1 Effects: For the first two overloads, let last2 be first2 + (last1 - first1). For each non-negative
integer n < min(last1 - first1, last2 - first2) performs: swap(*(first1 + n), *(first2 +

2 Requires: The two ranges [first1,last1) and [first2,last2) shall not overlap. *(first1 + n)
shall be swappable with (4.2.11) *(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.

7.4.4 Transform

template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&(projected<I, Proj>)>>()
tagged_pair<tag::in(I), tag::out(O)>
  transform(I first, S last, 0 result, F op, Proj proj = Proj{});

template <InputRange Rng, WeaklyIncrementable O, class F, class Proj = identity>
  requires Writable<O, indirect_result_of_t<F&amp;(projected<iterator_t<Rng>, Proj>)>>()
tagged_pair<tag::in(safe_iterator_t<Rng>), tag::out(O)>
  transform(Rng&& rng, O result, F op, Proj proj = Proj{});

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, WeaklyIncrementable O, class F, class Proj1 = identity, class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&amp;(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, 0 result,
    F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// A.2 (deprecated):
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, WeaklyIncrementable O, class F, class Proj1 = identity, class Proj2 = identity>
  requires Writable<O, indirect_result_of_t<F&amp;(projected<iterator_t<R>, Proj1>, projected<I, Proj2>)>>()
tagged_tuple<tag::in1(safe_iterator_t<R1>), tag::in2(I), tag::out(O)>
  transform(Rng&& rng, I first2, O result,
    F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

§ 7.4.4
template <InputRange Rng1, InputRange Rng2, WeaklyIncrementable O, class F,
         class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_of_t<F&(
    projected<iterator_t<Rng1>, Proj1>, projected<iterator_t<Rng2>, Proj2>)>>()
tagged_tuple<tag::in1(safe_iterator_t<Rng1>),
    tag::in2(safe_iterator_t<Rng2>),
    tag::out(0)>
transform(Rng1&& rng1, Rng2&& rng2, O result,
    F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

1 For binary transforms that do not take last2, let last2 be first2 + (last1 - first1). Let N be (last1 - first1) for unary transforms, or min(last1 - first1, last2 - first2) for binary transforms.

2 Effects: Assigns through every iterator \( i \) in the range \([\text{result}, \text{result} + N)\) a new corresponding value equal to \( \text{invoke(op, invoke(proj, *(first1 + (i - result))))} \) or \( \text{invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result))))} \).

3 Requires: \( \text{op} \) and \( \text{binary_op} \) shall not invalidate iterators or subranges, or modify elements in the ranges \([\text{first1}, \text{first1} + N]\), \([\text{first2}, \text{first2} + N]\), and \([\text{result}, \text{result} + N]\).\(^7\)

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 \( \text{op} \) or \( \text{binary_op} \).

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

### 7.4.5 Replace

[alg.replace]

template <ForwardIterator 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 <ForwardRange 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 <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectPredicate<projected<I, Proj>> Pred>
requires Writable<I, const T&>()
    I
    replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = Proj{});

template <ForwardRange Rng, class T, class Proj = identity,
    IndirectPredicate<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{});

\(^7\) The use of fully closed ranges is intentional.
Effects: Substitutes elements referred by the iterator \( i \) in the range \([\text{first}, \text{last})\) with \( \text{new\_value} \), when the following corresponding conditions hold: \( \text{invoke} (\text{proj}, *i) == \text{old\_value}, \text{invoke} (\text{pred}, \text{invoke} (\text{proj}, *i)) != \text{false} \).

Returns: \( \text{last} \).

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

```cpp
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());
```

```cpp
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());
```

```cpp
template <InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
class Proj = identity, IndirectPredicate<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());
```

```cpp
template <InputRange Rng, class T, OutputIterator<const T&> O, class Proj = identity,
IndirectPredicate<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 \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

5 Effects: Assigns to every iterator \( i \) in the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) either \( \text{new\_value} \) or \(*((\text{first} + (i - \text{result}))\) depending on whether the following corresponding conditions hold:

- \( \text{invoke} (\text{proj}, *((\text{first} + (i - \text{result})))) == \text{old\_value} \)
- \( \text{invoke} (\text{pred}, \text{invoke} (\text{proj}, *((\text{first} + (i - \text{result})))))) != \text{false} \)

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

7 Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate and projection.

7.4.6 Fill

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

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

template <class T, OutputIterator<const T&> O>
  O fill_n(O first, difference_type_t<O> n, const T& value);

Effects: fill assigns value through all the iterators in the range [first, last). fill_n assigns value through all the iterators in the range [first, first + n) if n is positive, otherwise it does nothing.

Returns: fill returns last. fill_n returns first + n for non-negative values of n and first for negative values.

Complexity: Exactly last - first, n, or 0 assignments, respectively.

7.4.7 Generate

[alg.generate]

template <Invocable F, OutputIterator<result_of_t<F&()>> O, Sentinel<O> S>
  O generate(O first, S last, F gen);

template <Invocable F, OutputRange<result_of_t<F&()>> Rng>
  safe_iterator_t<Rng> generate(Rng&& rng, F gen);

template <Invocable F, OutputIterator<result_of_t<F&()>> O>
  O generate_n(O first, difference_type_t<O> n, F gen);

Effects: Assigns the value of invoke(gen) through successive iterators in the range [first, last), where last is first + max(n, 0) for generate_n.

Returns: last.

Complexity: Exactly last - first evaluations of invoke(gen) and assignments.

7.4.8 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 prog = 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 prog = Proj());

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,>
  IndirectPredicate<projected<I, Proj>>() Pred>
  requires Permutable<I>()
  I remove_if(I first, S last, Pred pred, Proj prog = Proj());

template <ForwardRange Rng, class Proj = identity,>
  IndirectPredicate<projected<iterator_t<Rng>, Proj>>() Pred>
  requires Permutable<iterator_t<Rng>>()
  safe_iterator_t<Rng> remove_if(Rng&& rng, Pred pred, Proj prog = Proj());
Effects: Eliminates all the elements referred to by iterator \(i\) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions hold: \(\text{invoke}(\text{proj}, \ast i) = \text{value}\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)) \neq \text{false}\).

Returns: The end of the resulting range.

Remarks: Stable (ISO/IEC 14882:2014 §17.6.5.7).

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

Note: each element in the range \([\text{ret}, \text{last})\), where \(\text{ret}\) is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by moving from elements that were originally in that range.

```cpp
template <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, IndirectPredicate<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,
          IndirectPredicate<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{});
```

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

Effects: Copies all the elements referred to by the iterator \(i\) in the range \([\text{first}, \text{last})\) for which the following corresponding conditions do not hold: \(\text{invoke}(\text{proj}, \ast i) = \text{value}\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)) \neq \text{false}\).

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

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

Remarks: Stable (ISO/IEC 14882:2014 §17.6.5.7).

### 7.4.9 Unique

```
7.4.9 147

unique(I first, S last, R comp) = R{};
```
requires Permutable<iterator_t<Rng>>()
safe_iterator_t<Rng>
unique(Rng& rng, R comp = R{}, Proj proj = Proj{});

Effects: For a nonempty range, eliminates all but the first element from every consecutive group of
equivalent elements referred to by the iterator `i` in the range `[first + 1, last)` for which the following
conditions hold: `invoke(proj, *(i - 1)) == invoke(proj, *i)` or `invoke(pred, invoke(proj, *(i - 1)),
*invokes(proj, *i)) != false`.

Returns: The end of the resulting range.

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

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`.

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

Complexity: For nonempty ranges, exactly `last - first - 1` applications of the corresponding
direct and no more than twice as many applications of the projection.

7.4.10 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);

Effects: For each non-negative integer `i < (last - first)/2`, applies `iter_swap` to all pairs of iter-
ators `first + i`, `(last - i) - 1`.

Returns: `last`.

Complexity: Exactly `(last - first)/2` swaps.

§ 7.4.10 148
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);

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

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

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

Complexity: Exactly last - first assignments.

7.4.11 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);

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

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

Remarks: This is a left rotate.

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

Complexity: At most 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);

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

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

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

Complexity: Exactly last - first assignments.
7.4.12 Shuffle

```cpp
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);
```

```cpp
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**: Permutates the elements in the range \([\text{first}, \text{last})\) such that each possible permutation of those elements has equal probability of appearance.
2. **Complexity**: Exactly \((\text{last} - \text{first}) - 1\) swaps.
3. **Returns**: \(\text{last}\)
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.

7.4.13 Partitions

```cpp
template <InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectPredicate<projected<I, Proj>> Pred>
bool is_partitioned(I first, S last, Pred pred, Proj proj = Proj{});
```

```cpp
template <InputRange Rng, class Proj = identity,
    IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
bool
is_partitioned(Rng&& rng, Pred pred, Proj proj = Proj{});
```

1. **Returns**: true if \([\text{first}, \text{last})\) is empty or if \([\text{first}, \text{last})\) is partitioned by \(\text{pred}\) and \(\text{proj}\), i.e. if all iterators \(i\) for which \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *i)) \neq \text{false}\) come before those that do not, for every \(i\) in \([\text{first}, \text{last})\).
2. **Complexity**: Linear. At most \(\text{last} - \text{first}\) applications of \(\text{pred}\) and \(\text{proj}\).

```cpp
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectPredicate<projected<I, Proj>> Pred>
requires Permutable<I>()
I partition(I first, S last, Pred pred, Proj proj = Proj{});
```

```cpp
template <ForwardRange Rng, class Proj = identity,
    IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
requires Permutable<iterator_t<Rng>>()
safe_iterator_t<Rng>
p partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

3. **Effects**: Permutates the elements in the range \([\text{first}, \text{last})\) such that there exists an iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *j)) \neq \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *k)) \neq \text{false}\)
4. **Returns**: An iterator \(i\) such that for every iterator \(j\) in the range \([\text{first}, i)\) \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *j)) \neq \text{false}\), and for every iterator \(k\) in the range \([i, \text{last})\), \(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *k)) \neq \text{false}\).
Complexity: If \( I \) meets the requirements for a BidirectionalIterator, at most \( \frac{(last - first)}{2} \) 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,
         IndirectPredicate<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,
         IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
requires Permutable<iterator_t<Rng>>()
safe_iterator_t<Rng>
stable_partition(Rng&& rng, Pred pred, Proj proj = Proj{});
```

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 \([first, i)\) \( \text{invoke(pred, invoke(proj, *j))} \neq \text{false} \), and for every iterator \( k \) in the range \([i, last)\), \( \text{invoke(pred, invoke(proj, *k))} \) \neq \text{false} \). The relative order of the elements in both groups is preserved.

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

Complexity: At most \( (last - first) \cdot \log(last - first) \) 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, IndirectPredicate<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,
         IndirectPredicate<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{});
```

Requires: The input range shall not overlap with either of the output ranges.

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

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

Complexity: Exactly \( last - first \) applications of \( pred \) and \( proj \).

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
           IndirectPredicate<projected<I, Proj>> Pred>
I partition_point(I first, S last, Pred pred, Proj proj = Proj{});
```

```
template <ForwardRange Rng, class Proj = identity,
           IndirectPredicate<projected<iterator_t<Rng>, Proj>> Pred>
```
7.5 Sorting and related operations

All the operations in 7.5 take an optional binary callable predicate of type Comp that defaults to less<>. 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.

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.

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`.

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).

7.5.1 Sorting

7.5.1.1 sort

```cpp
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>, class Proj = identity>
requires Sortable<I, Comp, Proj>()
I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```cpp
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
safe_iterator_t<Rng>
sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

Effects: Sorts the elements in the range `[first, last)`.
Complexity: $O(N \log(N))$ (where $N == last - first$) comparisons.

7.5.1.2 stable_sort

```cpp
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>, class Proj = identity>
requires Sortable<I, Comp, Proj>()
I stable_sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```cpp
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
safe_iterator_t<Rng>
```

§ 7.5.1.2
stable_sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

**Effects:** Sorts the elements in the range `[first, last)`.

**Complexity:** It does at most \( N \log^2(N) \) (where \( N = last - first \)) comparisons; if enough extra memory is available, it is \( N \log(N) \).

**Remarks:** Stable (ISO/IEC 14882:2014 §17.6.5.7).

### 7.5.1.3 **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{});
```

**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.

**Complexity:** It takes approximately \( (last - first) \cdot \log(middle - first) \) comparisons.

### 7.5.1.4 **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{});
```

**Effects:** Places the first \( \min(last - first, result_last - result_first) \) sorted elements into the range `[result_first, result_first + \min(last - first, result_last - result_first))`.

**Returns:** The smaller of: `result_last` or `result_first + (last - first)`.

**Complexity:** Approximately \( (last - first) \cdot \log(\min(last - first, result_last - result_first)) \) comparisons.

### 7.5.1.5 **is_sorted**

```
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
          IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
```

§ 7.5.1.5
bool is_sorted(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
        IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>>, Comp = less<>>
bool
    is_sorted(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

    Returns: is_sorted_until(first, last, comp, proj) == last

template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
        IndirectStrictWeakOrder<projected<I, Proj>>, Comp = less<>>
I is_sorted_until(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity,
        IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>>, Comp = less<>>
safe_iterator_t<Rng>
    is_sorted_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

    Returns: If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in [first, last] for which the range [first, i) is sorted.

    Complexity: Linear.

### 7.5.2 Nth element

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
        class Proj = identity>
requires Sortable<I, Comp, Proj>()
I nth_element(I first, I nth, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
safe_iterator_t<Rng>
    nth_element(Rng&& rng, iterator_t<Rng> nth, Comp comp = Comp{}, Proj proj = Proj{});
```

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 Complexity: Linear on average.

### 7.5.3 Binary search

All of the algorithms in this section are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the 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.

#### 7.5.3.1 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{});
```

§ 7.5.3.1 154
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 furthestmost 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 $\log_2(last - first) + O(1)$ applications of the comparison function and projection.

7.5.3.2 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 furthestmost 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 $\log_2(last - first) + O(1)$ applications of the comparison function and projection.

7.5.3.3 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)}
Complexity: At most \(2 \times \log_2(last - first) + O(1)\) applications of the comparison function and projection.

### 7.5.3.4 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 \(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, e), value)\) and \(!\text{invoke}(\text{comp}, value, \text{invoke}(\text{proj}, e))\). Also, for all elements \(e\) of \([first, last)\), \(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, e), value)\) shall imply \(!\text{invoke}(\text{comp}, value, \text{invoke}(\text{proj}, e))\).

2. **Returns:** true if there is an iterator \(i\) in the range \([first, last)\) that satisfies the corresponding conditions: 
   \(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *i), value) == \text{false} \&\& \text{invoke}(\text{comp}, value, \text{invoke}(\text{proj}, *i)) == \text{false}\).

3. **Complexity:** At most \(\log_2(last - first) + O(1)\) applications of the comparison function and projection.

### 7.5.4 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, 0 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 \(result_last = result + (last1 - first1) + (last2 - first2)\). If an element \(a\) precedes \(b\) in an input range, \(a\) is copied into the output range before \(b\). If \(e_1\) is an element of \([first1, last1)\) and \(e_2\) of \([first2, last2)\), \(e_2\) is copied into the output range before \(e_1\) if and only if \(\text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj2}, e_2), \text{invoke}(\text{proj1}, e_1)))\) is true.

2. **Requires:** The ranges \([first1, last1)\) and \([first2, last2)\) shall be sorted with respect to \(\text{comp}\), \(\text{proj1}\), and \(\text{proj2}\). The resulting range shall not overlap with either of the original ranges.

3. **Returns:** \(\text{make_tagged_tuple}(\text{tag::in1}, \text{tag::in2}, \text{tag::out})(last1, last2, result_last)\).
Complexity: At most \((\text{last1} - \text{first1}) + (\text{last2} - \text{first2}) - 1\) applications of the comparison function and each projection.

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{});

Effects: Merges two sorted consecutive ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\), putting the result of the merge into the range \([\text{first}, \text{last})\). The resulting range will be in non-decreasing order; that is, for every iterator \(i\) in \([\text{first}, \text{last})\) other than \(\text{first}\), the condition \(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *i), \text{invoke}(\text{proj}, *(i - 1)))\) will be false.

Requires: The ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) shall be sorted with respect to \(\text{comp}\) and \(\text{proj}\).

Returns: last

Complexity: When enough additional memory is available, \((\text{last} - \text{first}) - 1\) applications of the comparison function and projection. If no additional memory is available, an algorithm with complexity \(\mathcal{O}\log(\mathcal{N})\) (where \(\mathcal{N}\) is equal to \(\text{last} - \text{first}\)) may be used.

Remarks: Stable (ISO/IEC 14882:2014 §17.6.5.7).

7.5.5 Set operations on sorted structures

This section defines all the basic set operations on sorted structures. They also work with multisets (ISO/IEC 14882:2014 §23.4.7) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to multisets in a standard way by defining \text{set\_union()} to contain the maximum number of occurrences of every element, \text{set\_intersection()} to contain the minimum, and so on.

7.5.5.1 includes

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
class Proj1 = identity, class Proj2 = identity,
IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<<>
bool
includes(I1 first1, S1 last1, I2 first2, S2 last2, Comp comp = Comp{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
class Proj2 = identity,
IndirectStrictWeakOrder<Projected<iterator_t<Rng1>, Proj1>,
Projected<iterator_t<Rng2>, Proj2>> Comp = less<<>
bool
includes(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Returns: true if \([\text{first2}, \text{last2})\) is empty or if every element in the range \([\text{first2}, \text{last2})\) is contained in the range \([\text{first1}, \text{last1})\). Returns false otherwise.

§ 7.5.5.1
### 7.5.5.2 `set_union`

```cpp
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
         WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
set_union(I1 first1, S1 last1, I2 first2, S2 last2, O result, Comp comp = Comp{},
          Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

**Effects:** Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

**Requires:** The resulting range shall not overlap with either of the original ranges.

**Returns:** The end of the constructed range.

**Complexity:** At most $2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ applications of the comparison function and projections.

**Remarks:** If `[first1, last1)` contains $m$ elements that are equivalent to each other and `[first2, last2)` contains $n$ elements that are equivalent to them, then all $m$ elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

### 7.5.5.3 `set_intersection`

```cpp
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
         WeaklyIncrementable O, class Comp = less<>, class Proj1 = identity, class Proj2 = identity>
requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>()
set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                 Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

**Effects:** Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

**Requires:** The resulting range shall not overlap with either of the original ranges.

**Returns:** The end of the constructed range.
Complexity: At most $2 \times ((\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)) - 1$ applications of the comparison function and projections.

Remarks: If $[\text{first}_1, \text{last}_1)$ contains $m$ elements that are equivalent to each other and $[\text{first}_2, \text{last}_2)$ 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.

7.5.5.4 set_difference

```
set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

Effects: Copies the elements of the range $[\text{first}_1, \text{last}_1)$ which are not present in the range $[\text{first}_2, \text{last}_2)$ to the range beginning at $\text{result}$. The elements in the constructed range are sorted.

Requires: The resulting range shall not overlap with either of the original ranges.

Returns: $\{\text{last}_1, \text{result} + n\}$, where $n$ is the number of elements in the constructed range.

Complexity: At most $2 \times ((\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)) - 1$ applications of the comparison function and projections.

Remarks: If $[\text{first}_1, \text{last}_1)$ contains $m$ elements that are equivalent to each other and $[\text{first}_2, \text{last}_2)$ contains $n$ elements that are equivalent to them, the last $\max(m - n, 0)$ elements from $[\text{first}_1, \text{last}_1)$ shall be copied to the output range.

7.5.5.5 set_symmetric_difference

```
set_symmetric_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
    Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});
```

Effects: Copies the elements of the range $[\text{first}_1, \text{last}_1)$ that are not present in the range $[\text{first}_2, \text{last}_2)$, and the elements of the range $[\text{first}_2, \text{last}_2)$ that are not present in the range $[\text{first}_1, \text{last}_1)$ to the range beginning at $\text{result}$. The elements in the constructed range are sorted.
Requires: The resulting range shall not overlap with either of the original ranges.

Returns: make_tagged_tuple<tag::in1, tag::in2, tag::out>(last1, last2, result + n), where n is the number of elements in the constructed range.

Complexity: At most 2 * ((last1 - first1) + (last2 - first2)) - 1 applications of the comparison function and projections.

Remarks: If [first1, last1) contains m elements that are equivalent to each other and [first2, last2) contains n elements that are equivalent to them, then |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.

7.5.6 Heap operations

A heap is a particular organization of elements in a range between two random access iterators [a, b). Its two key properties are:

1. There is no element greater than *a in the range and
2. *a may be removed by pop_heap(), or a new element added by push_heap(), in \( O(\log(N)) \) time.

These properties make heaps useful as priority queues.

make_heap() converts a range into a heap and sort_heap() turns a heap into a sorted sequence.

7.5.6.1 push_heap

```cpp
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{});
```

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.

7.5.6.2 pop_heap

```cpp
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>, class Proj = identity>
requires Sortable<I, Comp, Proj>()
I pop_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

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.
Returns: last

Complexity: At most $2 \times \log(last - first)$ applications of the comparison function and projection.

7.5.6.3 make_heap

```cpp
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>,
         class Proj = identity>
requires Sortable<I, Comp, Proj>()
I make_heap(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

```cpp
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
safe_iterator_t<Rng>
make_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

Effects: Constructs a heap out of the range [first,last).

Returns: last

Complexity: At most $3 \times (last - first)$ applications of the comparison function and projection.

7.5.6.4 sort_heap

```cpp
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{});
```

```cpp
template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
sort_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

Effects: Sorts elements in the heap [first,last).

Requires: The range [first,last) shall be a valid heap.

Returns: last

Complexity: At most $N \log(N)$ comparisons (where $N == last - first$).

7.5.6.5 is_heap

```cpp
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{});
```

```cpp
template <RandomAccessRange Rng, class Proj = identity,
          IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
bool
is_heap(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

Returns: is_heap_until(first, last, comp, proj) == last

```cpp
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{});
```

```cpp
template <RandomAccessRange Rng, class Proj = identity,
          IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less<>>
safe_iterator_t<Rng>
```

§ 7.5.6.5
is_heap_until(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Returns: If distance(first, last) < 2, returns last. Otherwise, returns the last iterator i in [first, last] for which the range [first, i) is a heap.

Complexity: Linear.

7.5.7 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{});

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

template <Copyable T, class Proj = identity,
          IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
constexpr T min(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

Requires: distance(rng) > 0.

Returns: The smallest value in the initializer_list or range.

Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the smallest.

template <class T, class Proj = identity,
          IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
constexpr const T& max(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});

Returns: The larger value.

Remarks: Returns the first argument when the arguments are equivalent.

template <Copyable T, class Proj = identity,
          IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
constexpr T max(initializer_list<T> rng, Comp comp = Comp{}, Proj proj = Proj{});

Requires: distance(rng) > 0.

Returns: The largest value in the initializer_list or range.

Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the largest.

template <class T, class Proj = identity,
          IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less<>>
constexpr tagged_pair<tag::min(const T&), tag::max(const T&)> minmax(const T& a, const T& b, Comp comp = Comp{}, Proj proj = Proj{});
Returns: \{b, a\} if b is smaller than a, and \{a, b\} otherwise.

Remarks: Returns \{a, b\} when the arguments are equivalent.

Complexity: Exactly one comparison and exactly two applications of the projection.

template <Copyable T, class Proj = identity, IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = less>
constexpr tagged_pair<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{});

Requires: distance(rng) > 0.

Returns: \{x, y\}, where x has the smallest and y has the largest value in the initializer_list or range.

Remarks: x is a copy of the leftmost argument when several arguments are equivalent to the smallest. y is a copy of the rightmost argument when several arguments are equivalent to the largest.

Complexity: At most $\frac{3}{2} \times$ distance(rng) applications of the corresponding predicate, and at most twice as many applications of the projection.

template <ForwardIterator I, Sentinel<I> S, class Proj = identity, IndirectStrictWeakOrder<projected<I, Proj>> Comp = less>
I min_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity, IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less>
safe_iterator_t<Rng>
min_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Returns: The first iterator i in the range [first, last) such that for every iterator j in the range [first, last) the following corresponding condition holds: invoke(comp, invoke(proj, *j), invoke(proj, *i)) == false. Returns last if first == last.

Complexity: Exactly max((last - first) - 1, 0) applications of the comparison function and exactly twice as many applications of the projection.

template <ForwardIterator I, Sentinel<I> S, class Proj = identity, IndirectStrictWeakOrder<projected<I, Proj>> Comp = less>
I max_element(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <ForwardRange Rng, class Proj = identity, IndirectStrictWeakOrder<projected<iterator_t<Rng>, Proj>> Comp = less>
safe_iterator_t<Rng>
max_element(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Returns: The first iterator i in the range [first, last) such that for every iterator j in the range [first, last) the following corresponding condition holds: invoke(comp, invoke(proj, *i), invoke(proj, *j)) == false. Returns last if first == last.

Complexity: Exactly max((last - first) - 1, 0) applications of the comparison function and exactly twice as many applications of the projection.
template <ForwardIterator I, Sentinel<I> S, class Proj = identity,
   IndirectStrictWeakOrder<projected<I, Proj>> Comp = less<>>
tagged_pair<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{});

Returns: \{first, first\} if \[first, last\) is empty, otherwise \{m, M\}, where m is the first iterator in \[first, last\) such that no iterator in the range refers to a smaller element, and where M is the last iterator in \[first, last\) such that no iterator in the range refers to a larger element.

Complexity: At most \(\max(\frac{3}{2}(N - 1)), 0\) applications of the comparison function and at most twice as many applications of the projection, where \(N\) is distance(first, last).

7.5.8 Lexicographical comparison

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
   class Proj1 = identity, class Proj2 = identity,
   IndirectStrictWeakOrder<projected<I1, Proj1>, projected<I2, Proj2>> Comp = less<>>
bool lexicographical_compare(I1 first1, S1 last1, I2 first2, S2 last2,
   Comp comp = Comp{}, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputRange Rng2, class Proj1 = identity,
   class Proj2 = identity,
   IndirectStrictWeakOrder<projected<iterator_t<Rng1>, Proj1>,
   projected<iterator_t<Rng2>, Proj2>> Comp = less<>>
bool lexicographical_compare(Rng1&& rng1, Rng2&& rng2, Comp comp = Comp{},
   Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

Returns: true if the sequence of elements defined by the range \[first1, last1\) is lexicographically less than the sequence of elements defined by the range \[first2, last2\) and false otherwise.

Complexity: At most \(2 \times \min((last1 - first1), (last2 - first2))\) applications of the corresponding comparison and projection.

Remarks: If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
  if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
  if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
}
return first1 == last1 && first2 != last2;

Remarks: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

7.5.9 Permutation generators

§ 7.5.9
template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
         class Proj = identity>
requires Sortable<I, Comp, Proj>()
bool next_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>,
         class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
bool
  next_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Effects: Takes a sequence defined by the range \([first, last)\) and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to \(\text{comp}\) and \(\text{proj}\). If such a permutation exists, it returns \(\text{true}\). Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns \(\text{false}\).

Complexity: At most \((last - first)/2\) swaps.

template <BidirectionalIterator I, Sentinel<I> S, class Comp = less<>,
         class Proj = identity>
requires Sortable<I, Comp, Proj>()
bool prev_permutation(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});

template <BidirectionalRange Rng, class Comp = less<>,
         class Proj = identity>
requires Sortable<iterator_t<Rng>, Comp, Proj>()
bool
  prev_permutation(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});

Effects: Takes a sequence defined by the range \([first, last)\) and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to \(\text{comp}\) and \(\text{proj}\).

Returns: \(\text{true}\) if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns \(\text{false}\).

Complexity: At most \((last - first)/2\) swaps.

### 7.6 C library algorithms

Table 10 describes some of the contents of the header \(<\text{cstdlib}\>\).

<table>
<thead>
<tr>
<th>Type</th>
<th>Name(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>size_t</td>
</tr>
<tr>
<td>Functions:</td>
<td>bsearch qsort</td>
</tr>
</tbody>
</table>

The contents are the same as the Standard C library header \(<\text{stdlib.h}\>\) with the following exceptions:

The function signature:

\[
\text{bsearch}(\text{const void *}, \text{const void *}, \text{size_t}, \text{size_t},
\text{int (*})(\text{const void *}, \text{const void *}))\]

is replaced by the two declarations:
extern "C" void* bsearch(const void* key, const void* base,
    size_t nmemb, size_t size,
    int (*compar)(const void*, const void*));

extern "C++" void* bsearch(const void* key, const void* base,
    size_t nmemb, size_t size,
    int (*compar)(const void*, const void*));

both of which have the same behavior as the original declaration.

The function signature:

    qsort(void *, size_t, size_t,
        int (*)(const void *, const void *));

is replaced by the two declarations:

    extern "C" void qsort(void* base, size_t nmemb, size_t size,
        int (*compar)(const void*, const void*));

    extern "C++" void qsort(void* base, size_t nmemb, size_t size,
        int (*compar)(const void*, const void*));

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by base are of trivial type.

[Note: Because the function argument compar() may throw an exception, bsearch() and qsort() are allowed to propagate the exception (ISO/IEC 14882:2014 §17.6.5.12). — end note]

See also: ISO C 7.10.5.
8 Numerics library

Header <experimental/ranges/random> synopsis

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    template <class G>
    concept bool UniformRandomNumberGenerator() { return see below; }
}}}}

8.1 Uniform random number generator requirements

// defined in <experimental/ranges/random>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    template <class G>
    concept bool UniformRandomNumberGenerator() {
        return requires(G g) {
            { g() } -> UnsignedIntegral; // not required to be equality preserving
            { G::min() } -> Same<result_of_t<G&()>>;
            { G::max() } -> Same<result_of_t<G&()>>;
        };
    }
}}}}

1 A uniform random number generator \( g \) of type \( G \) is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. \( \text{Note: The degree to which } g \text{'s results approximate the ideal is often determined statistically.} \) — end note

2 Let \( g \) be any object of type \( G \). Then \( \text{UniformRandomNumberGenerator}<G>() \) is satisfied if and only if

\[
\begin{align*}
\text{(2.1)} & \quad \text{Both } G::\text{min()} \text{ and } G::\text{max()} \text{ are constant expressions (ISO/IEC 14882:2014 §5.19)}. \\
\text{(2.2)} & \quad G::\text{min()} < G::\text{max}(). \\
\text{(2.3)} & \quad G::\text{min()} \leq g(). \\
\text{(2.4)} & \quad g() \leq G::\text{max}(). \\
\text{(2.5)} & \quad g() \text{ has amortized constant complexity.}
\end{align*}
\]
Annex A  (normative)
Compatibility features

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

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

1 The following algorithms are deemed unsafe and are deprecated in this document.

// 7.3.10, mismatch
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2,
class Proj1 = identity, class Proj2 = identity,
IndirectPredicate<projected<I1, Proj1>, projected<I2, Proj2>> Pred = equal_to<>
tagged_pair<tag::in1(I1), tag::in2(I2)>
mismatch(I1 first1, S1 last1, I2 first2, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputIterator I2,
class Proj1 = identity, class Proj2 = identity,
IndirectPredicate<projected<input_iterator_t<Rng1>, Proj1>,
projected<I2, Proj2>> Pred = equal_to>
tagged_pair<tag::in1(safe_iterator_t<Rng1>), tag::in2(I2)>
mismatch(Rng1 rng1, I2 first2, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 7.3.11, equal
template <InputIterator I1, Sentinel<I1> S1, InputIterator I2,
class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool equal(I1 first1, S1 last1,
I2 first2, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng1, InputIterator I2, class Pred = equal_to<>,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<input_iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool equal(Rng1 rng1, I2 first2, Pred pred = Pred{},
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 7.3.12, is_permutation

§ A.2
template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, 
class Pred = equal_to<>, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>()
bool is_permutation(I1 first1, S1 last1, I2 first2, 
Pred pred = Pred{}, 
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <ForwardRange Rng1, ForwardIterator I2, class Pred = equal_to<>, 
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<Rng1>, I2, Pred, Proj1, Proj2>()
bool is_permutation(Rng1&& rng1, I2 first2, Pred pred = Pred{}, 
Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

// 7.4.3, swap_ranges

template <ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2>
requires IndirectlySwappable<I1, I2>()
tagged_pair<tag::in1(I1), tag::in2(I2)>
swap_ranges(I1 first1, S1 last1, I2 first2);

template <ForwardRange Rng, ForwardIterator I>
requires IndirectlySwappable<iterator_t<Rng>, I>()
tagged_pair<tag::in1(safe_iterator_t<Rng>), tag::in2(I)>
swap_ranges(Rng&& rng1, I first2);

// 7.4.4, transform

template <InputIterator I1, Sentinel<I1> S1, InputIterator I2, WeaklyIncrementable O, 
class F, class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_of_t<F& (projected<I1, Proj1>, projected<I2, Proj2>)>>()
tagged_tuple<tag::in1(I1), tag::in2(I2), tag::out(O)>
transform(I1 first1, S1 last1, I2 first2, O result, 
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

template <InputRange Rng, InputIterator I, WeaklyIncrementable O, class F, 
class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_of_t<F& (projected<iterator_t<Rng>, Proj1>, projected<I, Proj2>)>>()
tagged_tuple<tag::in1(safe_iterator_t<Rng>), tag::in2(I), tag::out(O)>
transform(Rng&& rng1, I first2, O result, 
F binary_op, Proj1 proj1 = Proj1{}, Proj2 proj2 = Proj2{});

§ A.2 169
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 (1.2), which was developed by a large group of experts as a test of the expressive power of the idea of concepts.

The authors would like to thank Andrew Sutton for his work on the Concepts TS (1.2), for his help formalizing the ideas of the range-v3 library [2] on which this work is based, and for his review of this document.

Sean Parent has made major contributions to both the foundations and the wording of this paper.

Stephan T. Lavavej offered a careful review of much of this document, a non-trivial undertaking.

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

C.1 C++ and Ranges

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

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.

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.

Alternate designs that were less impactful were considered and dismissed. See Section 3.3.6 in N4128 (1.2) for a discussion of the issues.

C.1.2 Stronger Constraints

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<`.

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.

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

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.

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  

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 \texttt{std::optional}). This can weaken class invariants.

The guarantee of default-constructibility simplifies the implementation of much iterator- and range-based code that would otherwise need to wrap iterators in \texttt{std::optional}. But the needs of backward-compatibility, the extra complexity to iterator implementors, and the weakened invariants may prove to be too great a burden.

We may in fact go even farther and remove the requirement of default-constructibility from the \texttt{Semiregular} concept. Time and experience will give us guidance here.

C.1.5 iterator_traits cannot be specialized  

In this STL design, \texttt{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 \texttt{difference_type}, \texttt{value_type}, and \texttt{iterator_category}. The need for this traits balkanization is because the associated types belong to separate concepts: \texttt{difference_type} belongs to \texttt{WeaklyIncrementable}; \texttt{value_type} belongs to \texttt{Readable}; and \texttt{iterator_category} belongs to \texttt{InputIterator}.

This breakage is intentional and inherent in the decomposition of the iterator concepts established by The Palo Alto report (1.2).

C.2 Ranges and the Palo Alto TR (N3351)  

The Palo Alto report (1.2) presents a comprehensive design for the Standard Template Library constrained with concepts. It served both as a basis for the Concepts Lite language feature and for this document. However, this document diverges from the Palo Alto report in small ways. The differences are in the interests of backwards compatibility, 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  

In the design presented in this document, the type of a range’s end delimiter may differ from the iterator representing the range’s start position. The reasons for this change are described in N4128 (1.2). This causes a number of differences from the Palo Alto report:

(1.1) The algorithms get an additional constraint for the sentinel.
(1.2) The return types of the algorithms are changed as described above (C.1.1).
(1.3) Some algorithms have operational semantics that require them to know the physical end position (e.g., \texttt{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 complexity guarantee of any algorithm.

C.2.2 Invocables and Projections  

Adobe’s Source Libraries [1] pioneered the use of \textit{callables} and \textit{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

In the Palo Alto report, the WeaklyIncrementable concept has an associated type called DistanceType, and the RandomAccessIterator concepts adds another associated type called DifferenceType. The latter is required to be convertible to the former, but they are not required to be the same type. (DifferenceType is required to be a signed integral type, but DistanceType need not be signed.) Although sensible from a soundness point of view, the author of this paper feels this is potentially a rich source of confusion. This paper hews closer to the current standard by having only one associated type, DifferenceType, and requiring it to be signed.

C.2.4 Distance Primitive is O(1) for Random Access Iterators

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: \( \text{distance}(i, j) = j - i \). 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 (6.2.8). The SizedSentinel concept requires that \( b - a \) return the number of times \( a \) would have to be incremented to make it compare equal to \( b \). Any type purporting to be a random access iterator that fails to meet that requirement is by definition not a valid random access iterator.

C.2.5 Output Iterators

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

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 [3]). 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


Index

x C++ Standard, 2
“The Palo Alto”, 2
argument, 9
class
  base, 9
Concepts TS, 2
constant iterator, 41
conventions, 8
function
  virtual member, 9
library
  Ranges, 6, 9
multi-pass guarantee, 46
mutable iterator, 41
namespace, 9
requirements, 7
  iterator, 40
    uniform random number generator, 167
restriction, 9, 10
statement
  iteration, 5
swappable, 15
swappable with, 15
undefined behavior, 100
uniform random number generator
  requirements, 167
unspecified, 153

§ C.2.6 175
Index of library names

adjacent_find, 136
advance, 67, 94
all_of, 134
any_of, 134
Assignable, 14
back_insert_iterator, 73
    back_insert_iterator, 74
    constructor, 74
back_inserter, 74
base
    counted_iterator, 91
    move_iterator, 80
    reverse_iterator, 70
bidirectional_iterator_tag, 66
BidirectionalIterator, 46
binary_search, 156
Boolean, 16

Common, 13
common_iterator, 84
    common_iterator, 85
    constructor, 86
    destructor, 86
    operator!=, 88
    operator*, 86
    operator++, 87
    operator-, 88
    operator->, 86
    operator=, 86
    operator==, 87
dangling, 95
dangling, 95
get_unsafe, 95
default_sentinel, 88
DefaultConstructible, 20
DerivedFrom, 13
Destructible, 19
difference_type, 61
difference_type_t, 61
distance, 67
distance(R&& r), 110
empty, 66
equal, 138
    istreambuf_iterator, 102
equal_range, 155
equal_to, 27
    equal_to<>, 28
EqualityComparable, 17
<experimental/ranges/algorithm>, 113
<experimental/ranges/iterator>, 52
<experimental/ranges/utility>, 24
failed
    ostreambuf_iterator, 104
fill, 145
fill_n, 145
find, 135
find_end, 135
find_first_of, 136
§ C.2.6
find_if, 135
find_if_not, 135
for_each, 134
forward_iterator_tag, 66
ForwardIterator, 46
front_insert_iterator, 75
construct, 75
front_insert_iterator, 75
front_inserter, 76
generate, 146
generate_n, 146
get_unsafe
dangling, 95
greater, 27
greater<>., 28
greater_equal, 28
greater_equal<>., 29
identity, 29
includes, 157
Incrementable, 44
indirect_result_of, 48
IndirectInvocable, 48
IndirectlyComparable, 51
IndirectlyCopyable, 51
IndirectlyCopyableStorable, 51
IndirectlyMovable, 50
IndirectlyMovableStorable, 50
IndirectlySwappable, 51
IndirectPredicate, 48
IndirectRegularInvocable, 48
IndirectRelation, 48
IndirectStrictWeakOrder, 48
inplace_merge, 157
input_iterator_tag, 66
InputIterator, 45
insert_iterator, 76
construct, 76
insert_iterator, 76
insertor, 77
Integral, 14
Invocable, 21
is_heap, 161
is_heap_until, 161
is_nothrow_swappabe, 32
is_nothrow_swappable_with, 31
is_partitioned, 150
is_permutation, 139
is_sorted, 153
is_sorted_until, 154
is_swappable, 31
is_swappable_with, 31
istream_iterator, 96
constructor, 98
destructor, 98
operator! =, 99
operator*, 98
operator++, 98
operator->, 98
operator==, 98
istreambuf_iterator, 100
constructor, 102
operator++, 102
Iterator, 44
iterator_category, 62
iterator_category_t, 62
iterator_traits, 63
less, 28
less<>., 29
less_equal, 28
less_equal<>., 29
lexicographical_compare, 164
lower_bound, 154
make_counted_iterator, 94
make_heap, 161
make_move_iterator, 82
make_move_sentinel, 84
make_reverse_iterator, 73
make_tagged_pair, 37
make_tagged_tuple, 38
max, 162
max_element, 163
merge, 156
Mergeable, 52
min, 162
min_element, 163
minmax, 162, 163
minmax_element, 164
mismatch, 137
Movable, 21
move, 142
move_backward, 142
move_iterator, 78
base, 80
constructor, 79
move_iterator, 79
operator! =, 81
operator*, 80
operator+, 80, 82
operator++, 80
operator+=, 81
operator-, 81, 82
operator-=, 81
operator<, 81
operator<=, 81
operator=, 80
operator==, 81
operator>, 81
operator>=, 82
operator[], 81
move_sentinel, 82
constructor, 83
move_sentinel, 83
operator!+, 83
operator-, 84
operator=, 83
operator==, 83
MoveConstructible, 20

next, 67
next_permutation, 164
none_of, 134
not_equal_to, 27
not_equal_to<>, 28
nth_element, 154

operator!=
    common_iterator, 88
    counted_iterator, 93
    istream_iterator, 99
    istreambuf_iterator, 103
    move_iterator, 81
    move_sentinel, 83
    reverse_iterator, 72
    unreachable, 96
operator*
    back_insert_iterator, 74
    common_iterator, 86
    counted_iterator, 91
    front_insert_iterator, 75
    insert_iterator, 77
    istream_iterator, 98
    istreambuf_iterator, 102
    move_iterator, 80
    ostream_iterator, 100
    ostreambuf_iterator, 104
    reverse_iterator, 70
operator+
    counted_iterator, 92, 94
move_iterator, 80, 82
reverse_iterator, 71, 73
operator++
    back_insert_iterator, 74
    common_iterator, 87
    counted_iterator, 91
    front_insert_iterator, 76
    insert_iterator, 77
    istream_iterator, 98
    istreambuf_iterator, 102
    move_iterator, 80
    ostream_iterator, 100
    ostreambuf_iterator, 104
    reverse_iterator, 70
operator+=
    counted_iterator, 92
    move_iterator, 81
    reverse_iterator, 71
operator-
    common_iterator, 88
    counted_iterator, 92, 94
    move_iterator, 81, 82
    move_sentinel, 84
    reverse_iterator, 71, 73
operator-=
    counted_iterator, 92
    move_iterator, 81
    reverse_iterator, 71
operator->
    common_iterator, 86
    istream_iterator, 98
    reverse_iterator, 70
operator--
    counted_iterator, 92
    move_iterator, 80
    reverse_iterator, 71
operator<
    counted_iterator, 93
    move_iterator, 81
    reverse_iterator, 72
operator<=
    counted_iterator, 93
    move_iterator, 81
    reverse_iterator, 72
operator==
    reverse_iterator, 70
    back_insert_iterator, 74
    common_iterator, 86
    counted_iterator, 91
    front_insert_iterator, 75
    insert_iterator, 76, 77
move_iterator, 80
move_sentinel, 83
ostream_iterator, 100
ostreambuf_iterator, 104
tagged, 36
operator==
  common_iterator, 87
  counted_iterator, 93
  istream_iterator, 98
  istreambuf_iterator, 102
  move_iterator, 81
  move_sentinel, 83
  reverse_iterator, 72
  unreachable, 96
operator>
  counted_iterator, 94
  move_iterator, 81
  reverse_iterator, 72
operator>=
  counted_iterator, 94
  move_iterator, 82
  reverse_iterator, 72
operator[]
  counted_iterator, 93
  move_iterator, 81
  reverse_iterator, 71
ostream_iterator, 99
  constructor, 100
  destructor, 100
  operator*, 100
  operator++, 100
  operator=, 100
ostreambuf_iterator, 103
  constructor, 103, 104
output_iterator_tag, 66
OutputIterator, 46

partial_sort, 153
partial_sort_copy, 153
partition, 150
partition_copy, 151
partition_point, 151
Permutable, 52
pop_heap, 160
Predicate, 22
prev, 68
prev_permutation, 165
projected, 50
proxy
  istreambuf_iterator, 101
push_heap, 160
random_access_iterator_tag, 66
RandomAccessIterator, 47
Readable, 41
Regular, 21
RegularInvocable, 22
Relation, 22
remove, 146
remove_copy, 147
remove_copy_if, 147
remove_if, 146
replace, 144
replace_copy, 145
replace_copy_if, 145
replace_if, 144
reverse, 148
reverse_copy, 148
reverse_iterator, 68
  reverse_iterator, 70
  base, 70
  constructor, 70
  make_reverse_iterator non-member function, 73
  operator++, 70
  operator=, 71
rotate, 149
rotate_copy, 149
rvalue_reference_t, 62
Same, 12
search, 139
search_n, 140
Semiregular, 21
Sentinel, 44
set_difference, 159
set_intersection, 158
set_symmetric_difference, 159
set_union, 158
shuffle, 150
SignedIntegral, 14
SizedSentinel, 45
sort, 152
sort_heap, 161
Sortable, 52
stable_partition, 151
stable_sort, 152
StrictTotallyOrdered, 18
swap, 25
  tagged, 37
  tagged, 36
swap_ranges, 142
Swappable, 15

§ C.2.6
tagged, 34
  operator=, 36
  swap, 36
  tagged, 36
tagged_tuple
  make_tagged_tuple, 38
transform, 143
tuple_element, 37
tuple_size, 37

unique, 147
unique_copy, 148
unreachable, 95
  operator!=, 96
  operator==, 96
UnsignedIntegral, 14
upper_bound, 155

value_type_t, 42

WeaklyEqualityComparable, 17
WeaklyIncrementable, 43
Writable, 43

§ C.2.6